60-141-02 LECTURE 9: ABSTRACT DATA STRUCTURES

Edited by Dr. Mina Maleki

Outline

- Introduction
- Self-referential data structures
- Memory Allocation
- Concept of linked structures
- Linked lists
- Stacks and Queues
- Trees

Introduction

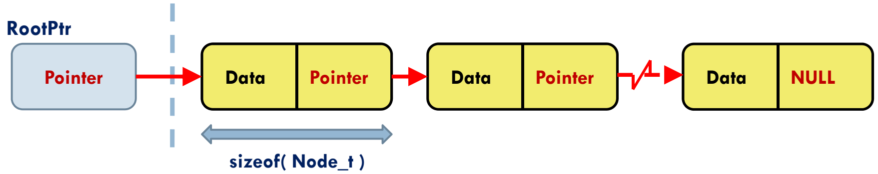

```
struct Nodetype { struct Nodetype * NextPtr ; …} ;
typedef struct Nodetype Node_t ;
Node_t * RootPtr ;
```

Memory block used for program variables with assigned names

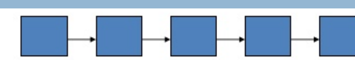
Memory block used for dynamically allocated blocks for storing data, each block addressable only using **pointers** (no names of variables!)

Statically allocated namespace

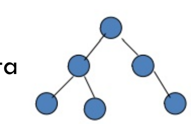
Dynamically allocated memory



Dynamic Data Structures

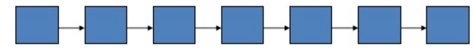
- Linked lists

  - Collections of data items "lined up in a row"
  - Insertions and deletions are made anywhere in a linked list.
- Stacks

  - Important in compilers and operating systems
  - Insertions and deletions are made only at one end of a stack—its top
- Queues

  - Represent waiting lines
  - Insertions are made only at the back (tail) of a queue and deletions are made only from the front (head) of a queue.
- Binary trees

  - Facilitate high-speed searching and sorting of data

Linked Lists

Linked lists



- A dynamic data structure whose length can be increased or decreased at run time.
- Contain pointer sub-fields, so that each pointer ***points at*** another allocated structure
- All linked lists must have an associated **root pointer (head)** that is a named pointer variable.
  - This provides the known address location to enter the list.
- There will be a **last**, or final, element and that one must have a **NULL** value in its link pointer to indicate the logical end-of-list.
- There are several kinds of linked list structures
  - Singly linked list
  - Doubly linked list

Basic Operations on a Linked List

1. Add a node.
2. Delete a node.
3. Search for a node.
4. Traverse (walk) the list.

Example 1

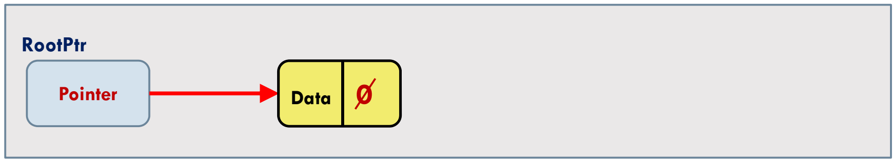
- Input data from a direct access file into a linked list.

```
typedef struct {
      int ID ;
      char Name[50] ;
      double Score ;
} Payload_t;

struct NodeStruct {
      Payload_t Data ;
      struct NodeStruct * NextPtr ;
} ;

typedef struct NodeStruct Node_t ;

Node_t * NewNodePtr, * Nptr, * RootPtr, * TempPtr ;
int PayloadSize = sizeof( Payload_t ),
int NodeSize = sizeof( Node_t ) ;
```

Example 1 …

- Always deal with the named root pointer first as a special case **– Head of the List**

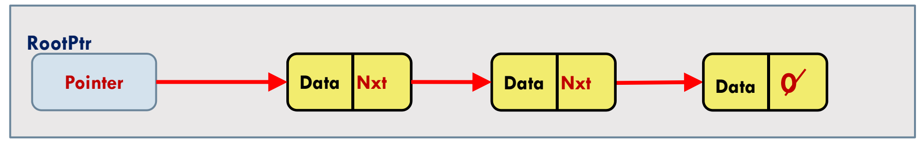
```
FILE * cfPtr ;
cfPtr = fopen( "DAFile.dat", "rb" ) ;

RootPtr = malloc(NodeSize) ; // NodeSize = sizeof( Node_t )
fread(RootPtr, PayloadSize, 1, cfPtr ) ;
RootPtr->NextPtr = NULL ;
```



Example 1 …

- And now deal with the dynamically allocated list

```
Nptr = RootPtr ;
while( ! feof( cfPtr )) {
    NewNodePtr = malloc( NodeSize ) ;
    fread(NewNodePtr, PayloadSize, 1, cfPtr ) ;
    NewNodePtr->NextPrt = NULL;

    Nptr->NextPtr = NewNodePtr;
    Nptr = NewNodePtr;
}
fclose( cfPtr ) ;
```



Example 2

- Traversal of a list to output data to stdout
  - Always start from the named root pointer

```
Nptr = RootPtr ; // Head of the list

while( Nptr != NULL ) {     // list traversal loop

   printf( "ID = %d, Name = %s, Score = %lf\n",
         Nptr->Data.ID, Nptr->Data.Name, Nptr->Data.Score);

   Nptr = Nptr->NextPtr ; //point at successor (next) element
}
```

Example 3

- Traversal of a list to find an ID (IDval)
  - This is a sequential search with complexity O(N) for a list with N nodes

```
Nptr = RootPtr ;        // Head of the list

while( Nptr != NULL) ) {    // list traversal loop

    if( Nptr->Data.ID == IDval ) // search criterion
      break ;

    Nptr = NPtr->NextPtr ;   //point at successor (next) element
}

if( Nptr != NULL )
    printf( "ID = %d, Name = %s, Score = %lf\n",
        Nptr->Data.ID, Nptr->Data.Name, Nptr->Data.Score);
else
    printf( "ID: %d not found\n", IDval ) ;
```

Example 4

- Deletion of a list
  - Also called *de-allocation* of memory
  - This is an extremely important issue for programmers
  - For every malloc() call there must be a matching free() call !

```
Nptr = RootPtr ;

while( Nptr != NULL) ) {        // Traverse the list
    TempPtr = Nptr ;            // 1. Save current node address
    Nptr = TempPtr->NextPtr ;   //  2. Point at successor node
    free(TempPtr) ;             //  3.  Release memory for node
}
RootPtr = NULL ;      // list is now empty!
```

CAUTION: Order of statements is important

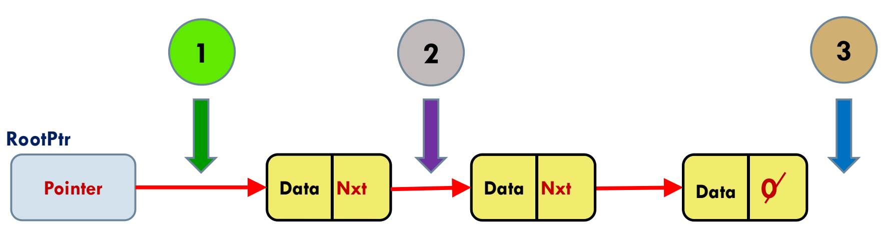
Example 5

- Input data into a linked list so that it is sorted with each insertion of a node – **Insertion Sort**
  - Recall from a previous example:

```
typedef struct {
      int ID ;
      char Name[50] ;
      double Score ; } Payload_t ;
struct NodeStruct {
      Payload_t Data ;
      struct NodeStruct * NextPtr ; } ;

typedef struct NodeStruct Node_t ;

FILE * cfPtr = fopen( “DAFile.dat”, “rb” ) ; ;
Node_t * NewNodePtr, * Nptr, * RootPtr ;
int PayloadSize = sizeof( Payload_t );
Int NodeSize = sizeof( Node_t ) ;
```

Example 5 …

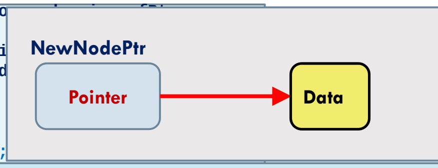
- We must consider three different special cases
  - **1. Insert at Head of List**
  - **2. Insert between two list nodes**
  - **3. Insert at End of List**




```
// ASSUMPTION: File has been opened using cfPtr
while( ! feof( cfPtr ) ) {
    NewNodePtr= malloc( NodeSize ) ;
    fread( NewNodePtr, PayloadSize, 1, cfPtr ) ;

    if( RootPtr == NULL ) {
        NewNodePtr->NextPtr = NULL ;
        RootPtr = NewNodePtr ;
    }
    else if(NewNodePtr->Data.ID < RootPtr->Data.ID ){
        NewNodePtr->NextPtr = RootPtr ;
        RootPtr = NewNodePtr ;
    }
    else {
        Node_t * PrevNodePtr = RootPtr ;
        Nptr = RootPtr->NextPtr ;

        while( Nptr != NULL ) {
            if( NewNodePtr->Data.ID < NPtr->Data.ID ){
                NewNodePtr->NextPtr = PrevNodePtr ;
                PrevNodePtr->NextPtr = NewNodePtr ;
                break ;
            }
            else {
                PrevNodePtr = Nptr ;
                Nptr = Nptr->NextPtr ;
            }
        }
        if( PrevNodePtr->NextPtr == NULL ) {
            NewNodePtr->NextPtr = NULL  ;
            PrevNodePtr->NextPtr = NewNodePtr ;
        }
    }
}
```

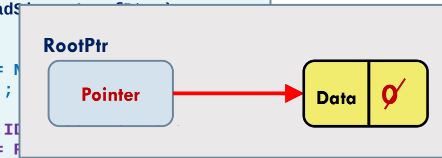
- Input data from a direct access file into a linked list
  - Here is a complete listing of the C code for this algorithm

```
// ASSUMPTION: File has been o
while( ! feof( cfPtr ) ) {
    NewNodePtr= malloc( NodeSi
    fread( NewNodePtr, Payload

    if( RootPtr == NULL ) {
        NewNodePtr->NextPtr =
        RootPtr = NewNodePtr ;
    }
    else if(NewNodePt
        NewNodePtr->N
        RootPtr = New
    }
    else {
        Node_t * Prev
        Nptr = RootPt

        while( Nptr !
            if( NewNo
                NewNo
                PrevN
                break
            }
            else {
                PrevNodePtr = Nptr ;
                Nptr = Nptr->NextPtr ;
            }
        }
        if( PrevNodePtr->NextPtr == NULL ) {
            NewNodePtr->NextPtr = NULL  ;
            PrevNodePtr->NextPtr = NewNodePtr ;
        }
    }
}
```



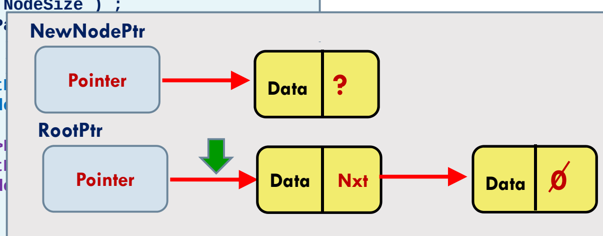

```
// ASSUMPTION: File has been opened using cfPtr
while( ! feof( cfPtr ) ) {
    NewNodePtr = malloc( NodeSize ) ;
    fread( NewNodePtr, PayloadSize, 1, cfPtr ) ;

   // Logic to find where to insert this inputted
   // data block  in the singly linked list.

}
```

```
// ASSUMPTION: File has been opened using cfPtr
while( ! feof( cfPtr ) ) {
    NewNodePtr= malloc( NodeSize ) ;
    fread( NewNodePtr, PayloadS

    if( RootPtr == NULL ) {
        NewNodePtr->NextPtr = N
        RootPtr = NewNodePtr ;
    }
    else if(NewNodePtr->Data.ID
        NewNodePtr->NextPtr = R
        RootPtr = NewNodePtr ;
    }
    else {
        Node_t * Prev
        Nptr = RootPt

        while( Nptr !
            if( NewNo
                NewNo
                PrevN
                break
            }
            else {
                PrevNodePtr = Nptr ;
                Nptr = Nptr->NextPtr ;
            }
        }
        if( PrevNodePtr->NextPtr == NULL ) {
            NewNodePtr->NextPtr = NULL  ;
            PrevNodePtr->NextPtr = NewNodePtr ;
        }
    }
}
```



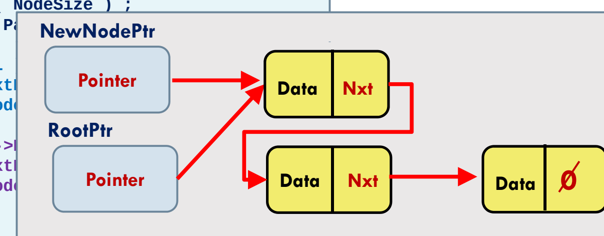

```
if( RootPtr == NULL ) {
   NewNodePtr->NextPtr = NULL ;
   RootPtr = NewNodePtr ;
}
```

```
// ASSUMPTION: File has been opened using cfPtr
while( ! feof( cfPtr ) ) {
    NewNodePtr= malloc( NodeSize ) ;
    fread( NewNodePtr, Pa

    if( RootPtr == NULL
        NewNodePtr->Next
        RootPtr = NewNode
    }
    else if(NewNodePtr->
        NewNodePtr->Next
        RootPtr = NewNode
    }
    else {
        Node_t * PrevNodePtr = RootPtr ;
        Nptr = RootPtr->NextPtr ;

        while( Np
            if( N
                N
                P
                b
            }
            else
                P
                N
            }
        }
        if( PrevN
            NewNodePtr->NextPtr = NULL  ;
            PrevNodePtr->NextPtr = NewNodePtr ;
        }
    }
}
```

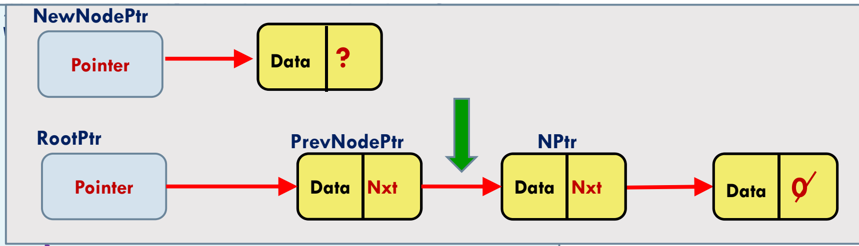



```
else if( NewNodePtr->Data.ID < RootPtr->Data.ID){
   NewNodePtr->NextPtr = RootPtr ;
   RootPtr = NewNodePtr ;
}
else {
   //  Locate insertion point in SL list
}
```

```
// ASSUMPTION: File has been opened using cfPtr
while( ! feof( cfPtr ) ) {
    NewNodePtr= malloc( NodeSize ) ;
    fread( NewNodePtr, Pa

    if( RootPtr == NULL
        NewNodePtr->Next
        RootPtr = NewNode
    }
    else if(NewNodePtr->
        NewNodePtr->Next
        RootPtr = NewNode
    }
    else {
        Node_t * PrevNodePtr = RootPtr ;
        Nptr = RootPtr->NextPtr ;

        while( Np
            if( N
                N
                P
                b
            }
            else
                P
                N
            }
        }
        if( PrevN
            NewNodePtr->NextPtr = NULL  ;
            PrevNodePtr->NextPtr = NewNodePtr ;
        }
    }
}
```

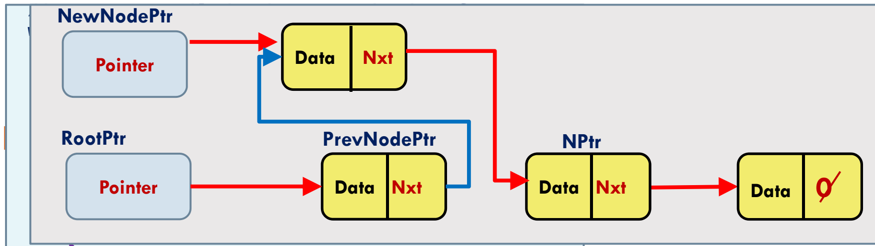



```
else if( NewNodePtr->Data.ID < RootPtr->Data.ID){
   NewNodePtr->NextPtr = RootPtr ;
   RootPtr = NewNodePtr ;
}
else {
   //  Locate insertion point in SL list
}
```



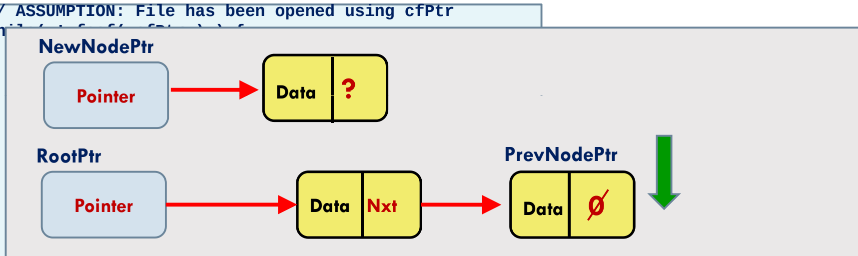

```
else {
    Node_t * P
    Nptr = Roo

    while( Npt
        if( Ne
```

```
Node_t * PrevNodePtr = RootPtr ; //previous node
Nptr = RootPtr->NextPtr ;  //current node

while( Nptr != NULL ) {
   if( NewNodePtr->Data.ID < NPtr->Data.ID ) {
      NewNodePtr->NextPtr = NPtr ;
      PrevNodePtr->NextPtr = NewNodePtr ;
      break ;
   }
    else {
       PrevNodePtr = Nptr ;
       Nptr = Nptr->NextPtr ;
    }
}
```




```
Node_t * PrevNodePtr = RootPtr ; //previous node
Nptr = RootPtr->NextPtr ;  //current node

while( Nptr != NULL ) {
   if( NewNodePtr->Data.ID < NPtr->Data.ID ) {
      NewNodePtr->NextPtr = NPtr ;
      PrevNodePtr->NextPtr = NewNodePtr ;
      break ;
   }
    else {
       PrevNodePtr = Nptr ;
       Nptr = Nptr->NextPtr ;
    }
}
```

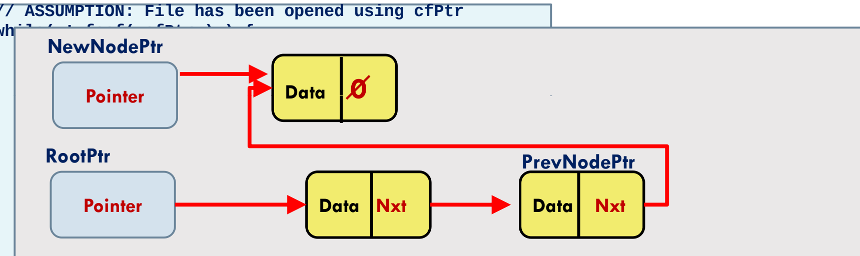
```
// ASSUMPTION: File has been opened using cfPtr
```




```
else {
    Node_t * PrevNodePtr = RootPtr ;
    Nptr = Roo
    ...
            PrevNodePtr = Nptr ;
            Nptr = Nptr->NextPtr ;
        }
    }
    if( PrevNodePtr->NextPtr == NULL ) {
        NewNodePtr->NextPtr = NULL  ;
        PrevNodePtr->NextPtr = NewNodePtr ;
    }
}
}
```

```
if( PrevNodePtr->NextPtr == NULL ) {
    NewNodePtr->NextPtr = NULL  ;
    PrevNodePtr->NextPtr = NewNodePtr ;
}
```

```
// ASSUMPTION: File has been opened using cfPtr
```




```
if( PrevNodePtr->NextPtr == NULL ) {
    NewNodePtr->NextPtr = NULL  ;
    PrevNodePtr->NextPtr = NewNodePtr ;
}
```

```
// ASSUMPTION: File has been opened using cfPtr
while( ! feof( cfPtr ) ) {
    NewNodePtr= malloc( NodeSize ) ;
    fread( NewNodePtr, PayloadSize, 1, cfPtr ) ;

    if( RootPtr == NULL
        NewNodePtr->Next
        RootPtr = NewNod
    }
    else if(NewNodePtr->
        NewNodePtr->Next
        RootPtr = NewNod
    }
    else {
        Node_t * PrevNod
        Nptr = RootPtr->

        while( Nptr != N
            if( NewNodeP
                NewNodeP
                PrevNode
                break ;
            }
            else {
                PrevNode
                Nptr = Nptr->NextPtr ;
            }
        }
        if( PrevNodePtr->NextPtr == NULL ) {
            NewNodePtr->NextPtr = NULL  ;
            PrevNodePtr->NextPtr = NewNodePtr ;
        }
    }
}
```

```
// USING LINKED LIST FUNCTIONS

// ASSUMPTION: File has been opened using cfPtr
while( ! feof( cfPtr ) ) {
    NewNodePtr = malloc( NodeSize ) ;
    fread( NewNodePtr, PayloadSize, 1, cfPtr ) ;
    NewNodePtr->NextPtr = NULL ;

    Nptr = FindNode ( &RootPtr, NewNodePtr->Data.ID ) ;

    InsertNode( &RootPtr, &Nptr, &NewNodePtr ) ;

}

// This still leaves  the functions to write, but it makes
//      it easier to understand the primary algorithm,
//      and the start of the Top-Down design process.
```

Linked list Functions

26

- Some other example functions to consider:
  - `Node_t * FindNode ( Node_t * Head, int IDval ) ;`
  - `void * InsertNode ( Node_t * Root,`
    `Node_t * InsertPtr,`
    `Node_t * NewNodePtr ) ;`
  - `Node_t * FindLast ( Node_t * Head ) ;`
  - `void * OutputList ( Node_t * Head ) ;`
  - `void * DeleteList ( Node_t * Head ) ;`
  - `int  isEmpty( Node_t * Head ) ;`
  - `long int  ListLength( Node_t * Head ) ;`

Linked List variations

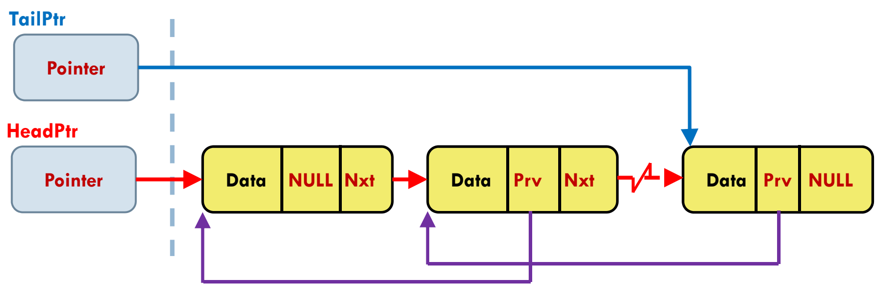
27

- Using linked lists, and by changing the way we carry out operations (such as insertion, deletion and traversal of nodes) we can define other abstract concepts such as:
  - doubly linked list
  - stack
  - queue
  - circular queue
  - tree
  - etc…

Doubly Linked List

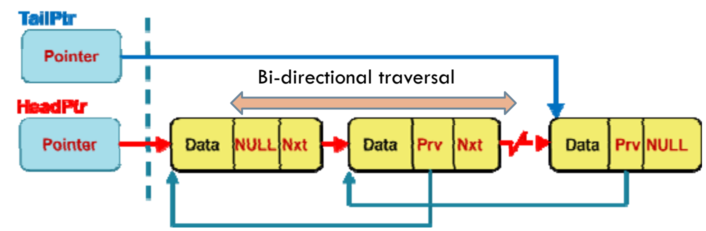
Doubly linked lists–Conceptual View

```
struct NodetypeDL {
       Payload_t   Data ;
       struct NodetypeDL * PrevPtr ;
       struct NodetypeDL * NextPtr ;
} ;
typedef struct NodetypeDL Node_tDL ;

Node_tDL * HeadPtr = NULL, * TailPtr = NULL ;
```



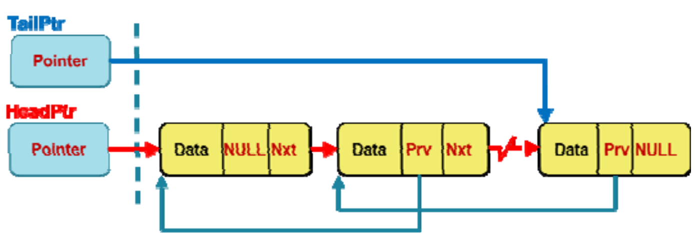
Doubly linked lists

- Doubly linked lists feature **two self-referential pointers**, usually called Predecessor (Previous) and Successor (Next) links
- There are two named pointers, usually called Head and Tail pointers, the latter pointing to the last node in the list
  - Traversal can be performed in both directions
- Typical operations are similar to those for singly linked lists
  - **`InsertNode`**, **`DeleteNode`**, **`DeleteList`**, **`FindNode`**, and so on



Doubly linked lists – Useful functions

- InsertHeadNode
- InsertTailNode
- FindNodebyID
- InsertNodebyID
- DeleteNodebyID
- DeleteList



InsertHeadNode

- Assume that `Nptr` points to the data structure to be inserted at the **head** of the list, and it is fully initialized, including both `Next` and `Prev` pointers with NULL values.

```
void InsertHeadNode( Node_tDL * Hptr,
                     Node_tDL * Tptr,
                     Node_tDL   Nptr ) {

    if( *Hptr == NULL )       // empty list, update tail
        *Tptr = Nptr ;
    else                      // update new node to point
        Nptr->NextPtr = *Hptr ;   // to rest of list

    *Hptr = Nptr ;               // update head and exit
    return ;     // success
}
```

Both head and tail pointer arguments can be modified (use call-by-reference).

InsertTailNode

- Assume that Nptr points to the data structure to be inserted at the **tail** of the list, and it is fully initialized, including both Next and Prev pointers with NULL values.

```
void InsertTailNode( Node_tDL * Hptr,
                     Node_tDL * Tptr,
                     Node_tDL   Nptr ) {

    if( *Tptr == NULL )        // empty list, update head
          *Hptr = Nptr ;
    else                           // update new node to point
          Nptr->PrevPtr = *Tptr ;  // to rest of list

    *Tptr = Nptr ;                 // update tail and exit
    return ;     // success
}
```

Both head and tail pointer arguments can be modified (use call-by-reference).

FindNodebyID

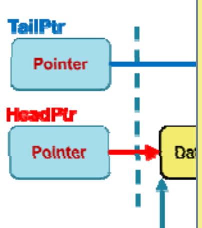
- Assume that IDval contains the search value and Hptr points to the head of the list.
- Returns a pointer to the first node containing IDval, otherwise returns NULL.

```
Node_tDL * FindNodebyID( Node_tDL Hptr, int  IDval ) {
   Node_tDL  Nptr ;
   if( *Hptr == NULL )     // empty list
        return NULL ;
   else {                           // search list
       Nptr = Hptr ;
       while( Nptr != NULL ) {
          if( Nptr->Data.ID == IDval )
              return Npr ;       //  search successful
          Nptr = Nptr->NextPtr ;
       }
   }
     return NULL ;
}
```

Doubly linked lists – Useful functions

- InsertHeadNode
- InsertTailNode
- **FindNodebyID**
- InsertNodebyID
- DeleteNodebyID
- DeleteList

```
// Assume that IDval contains the search value and
// Hptr points to the head of the list.
// Returns a pointer to the first node containing IDval,
// otherwise returns NULL.

Node_tDL * FindNodebyID( Node_tDL Hptr, int  IDval ) {
    Node_tDL  Nptr ;

    if( *Hptr == NULL )     // empty list
        return NULL ;
    else {                          // search list
        Nptr = Hptr ;
        while( Nptr != NULL ) {
            if( Nptr->Data.ID == Idval )
                return Npr ;      //  search successful
            Nptr = Nptr->NextPtr ;
        }
    }
    return NULL ;
}
```



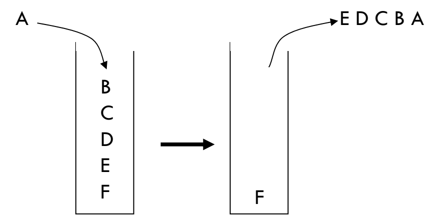
Stacks

Stacks and Queues

- There are two kinds of **singly linked list** structures that merit special attention
- A **Stack** is a linked list with an access pointer called the Stack Pointer, and whose nodes are added or deleted only at the beginning of the list
  - They are referred to as **LIFO** (Last In, First Out) lists
- A **Queue** is a linked list with two access pointers, called Head and Tail, and whose nodes are added only to the Tail, or deleted only from the Head of the list
  - They are referred to as **FIFO** lists (First In, First Out)

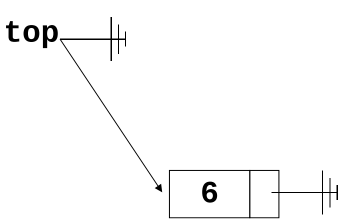
Stacks

- A Stack is a linked list with an access pointer called the Stack (or top) Pointer
- Nodes are added or deleted only at the beginning (top) of the list
- They are referred to as LIFO (Last In, First Out) data structures



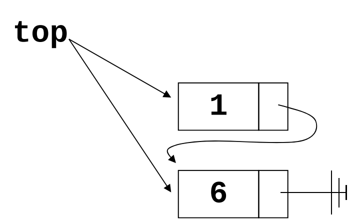
Stack Example


```
push(&top,6);
```



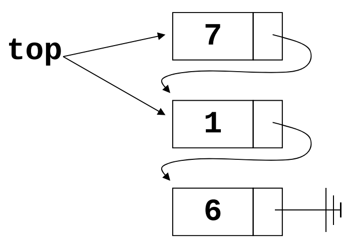
Stack Example


```
push(&top,6);
push(&top,1);
```



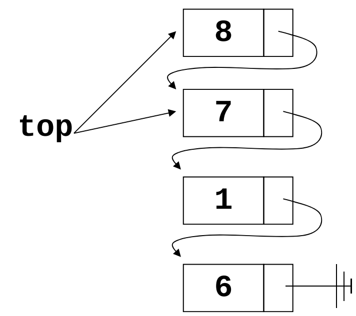
Stack Example


```
push(&top,6);
push(&top,1);
push(&top,7);
```



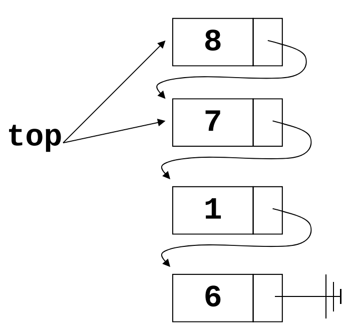
Stack Example


```
push(&top,6);
push(&top,1);
push(&top,7);
push(&top,8);
```



Stack Example


```
push(&top,6);
push(&top,1);
push(&top,7);
push(&top,8);
pop(&top);
```

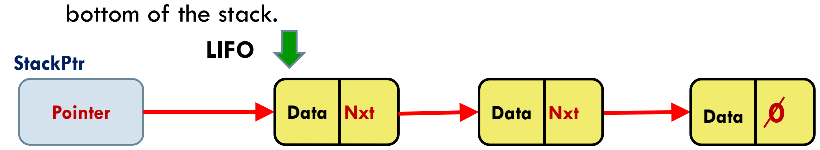


Stack Operations

- PUSH – **insert** a new node at the **top of the stack**
- POP – **remove** (delete) a new node from the **top of the stack**
- The named entry pointer is typically called the Stack/top Pointer

```
struct stackNode {
   Data_t Data; //Data_t can be any types such as int,char,a struct or …
   struct stackNode * Nextptr ;
}
typedef struct stackNode StackNode ;
typedef StackNode * StackNodePtr ;
StackNodePtr top;  //Points to stacktop
```

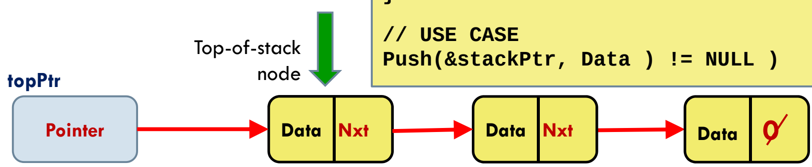
- The NextPtr of the last node of the satck is set to Null to indicate the bottom of the stack.



Stacks – Push()

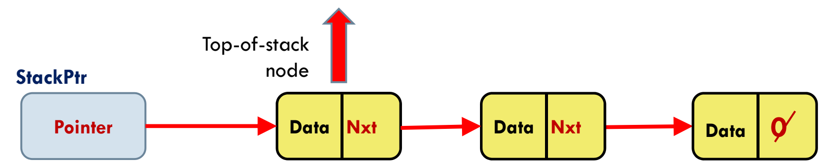
- Push() inserts a node into the first list position of the stack
- The new top-of-stack node must point to the rest of the list
- The stack pointer must point at the new top-of-stack node

```
void Push(StackNodePtr * topPtr,
          Data_t D) {

   StackNodePtr NewPtr ;
   NewPtr = malloc(sizeof(StackNode));

   if( NewPtr != NULL ){
      Copy(NewPtr->Data, D);
      NewPtr->NextPtr = * topPtr;
      * topPtr = NewPtr ;
   }
   else
      printf( “ No memory available!”);
}

// USE CASE
Push(&stackPtr, Data ) != NULL )
```



Stacks – Pop()

- Pop( ) obtains and returns the data from the top-of-stack node, then it removes this node from the stack
- A return mechanism must be chosen and implemented for the data
  - Scalar data can be returned directly through function return statements – this limits the ability to report errors such as an empty stack
  - Call-by-reference is always useful, but necessary for complex data structures
- The stack pointer must be updated to point at the second node, which then becomes the new top-of-stack node
- The used node is then de-allocated (and its memory block can be re-used)



Stacks – Pop() …


```
int Pop( StackNodePtr * topPtr, Data_t * Dptr ) {
   StackNodePtr tempPtr ;
   Data_t D ;

   if( *topPtr == NULL ) // Stack is empty so fail
     return 0 ;
   tempPtr = *topPtr;

   Copy(Dptr,(* topPtr)->Data) ;  // Copy-return payload data

   topPtr= (* topPtr)->NextPtr ;     // Get address of next node

   free(tempPtr ) ;                  // De-allocate top node
   return 1 ;                        // Success is TRUE
}

// USE CASE :: Assume:  Data_t DataRec ;StackNodePtr * StackPtr ;
if( Pop( &StackPtr, &DataRec ) )
     printf( “ID = %d\n”, DataRec.ID ) ;
else
     printf( “No data found on empty stack\n” ) ;
```

Applications of Stacks

- Stacks have many interesting applications.
- For example, whenever a function call is made, the called function must know how to return to its caller, so the return address is pushed onto a stack.
- If a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order so that each function can return to its caller.
- Stacks support recursive function calls in the same manner as conventional nonrecursive calls.
- When the function returns to its caller, the space for that function's automatic variables is popped off the stack, and these variables no longer are known to the program.
- Stacks are used by compilers in the process of evaluating expressions and generating machine-language code.

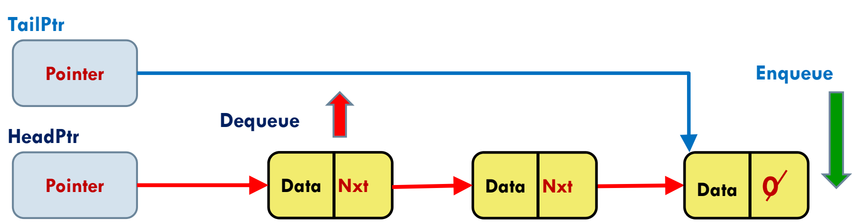
Queues

Queues

- A Queue is a linked list with two access pointers, called Head and Tail
- Nodes are added only to the Tail
- Nodes are deleted only from the Head of the list
- They are referred to as FIFO lists (First In, First Out)
-
```
struct queueNode {
    Data_t   Data ;  //Data_t can be any types such as int, char, a struct or …
    struct queueNode * Nextptr ;
}
typedef struct queueNode QueueNode ;
typedef QueueNode * QueueNodePtr ;
```

Queue Operations

- enqueue() – insert a node at the Tail position
- dequeue() – obtain data from the node at the Head position, then delete the node

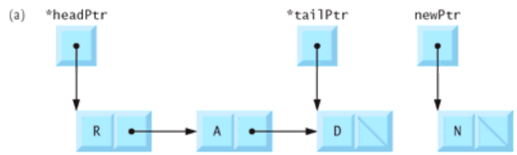


Queues- enqueue()

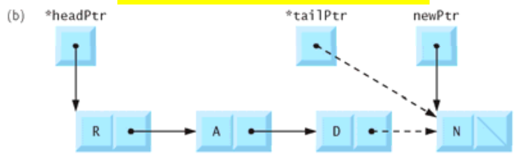

```
void enqueue( QueueNodePtr * headPtr,
              QueueNodePtr * tailPtr, Data_t D ) {
   QueueNodePtr newPtr ;
   newPtr= malloc( sizeof(QueueNode) ) ;

   if(newPtr != NULL )      {
      Copy(newPtr->Data, D ) ;
      newPtr->NextPtr = NULL ;

        if( *headPtr == NULL )   //empty queue
           *headPtr = newPtr;
        else
           (*tailPtr)->NextPtr = newPtr;

        *tailPtr= newPtr;
   }
   else
        printf(“No memory available”);
}
```

Example

enqueue() – insert a node at the Tail position



`*tailPtr= newPtr; //2`



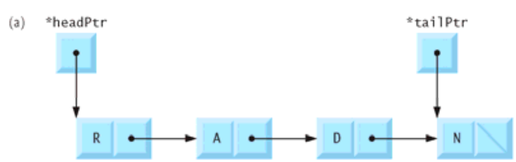
`(*tailPtr)->NextPtr = newPtr; //1`

Queues- dequeue()

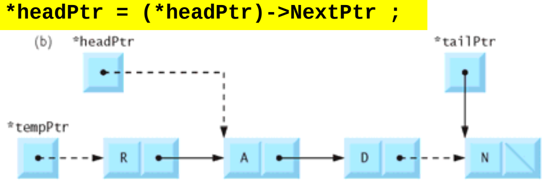

```
int dequeue(QueueNodePtr * headPtr, Data_t * Dptr ) {
   QueueNodePtr tempPtr ;
   Data_t D ;

   if( *headPtr == NULL ) // Queue is empty so fail
       return 0 ;

   tempPtr = *headPtr;              // Get address of next node

   Copy((Dptr,(*headPtr)->Data) ;    // Copy-return payload data

   *headPtr = (*headPtr)->NextPtr ; // Update head pointer

   free(tempPtr) ;    // Deallocate node
   return 1 ;           // Success is TRUE
}
```

Example

dequeue() – obtain data from the node at the Head position

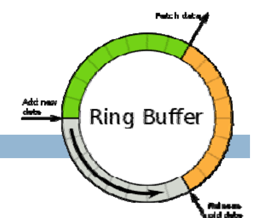


`*headPtr = (*headPtr)->NextPtr ;`

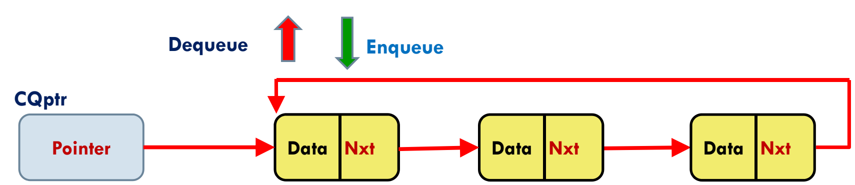


`tempPtr = *headPtr;`

Circular Queues



- There are different kinds of queues
- **Circular Queues** are singly linked lists with only a single named pointer, sometimes called **`CQptr`**
  - All enqueue and dequeue operations are performed using **`CQptr`**
  - **`CQptr == NULL`** indicates an empty queue
  - A further distinction is that the last node points to the first node – the **`NextPtr`** field is never **`NULL`**!
    - Not necessarily FIFO
    - Sometimes called Round-Robin queue



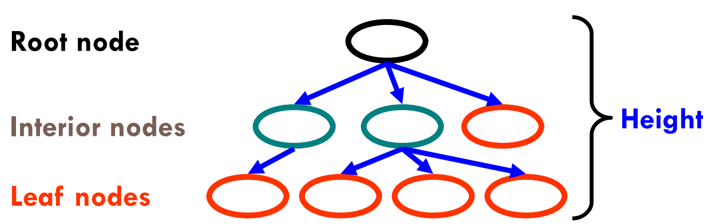
Applications of Queues

- For computers that have only a single processor, only one user at a time may be serviced.
  - Entries for the other users are placed in a queue.
  - The entry at the front of the queue is the next to receive service.
- Queues are also used to support print spooling.
  - A multiuser environment may have only a single printer.
  - If the printer is busy, other outputs may still be generated.
- Information packets also wait in queues in computer networks.
  - Each time a packet arrives at a network node, it must be routed to the next node on the network along the path to its final destination.
  - The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.

Trees

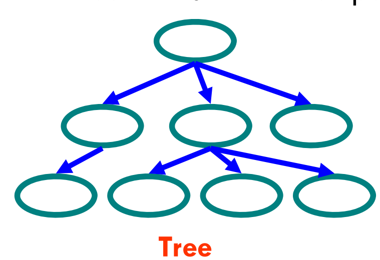
Trees

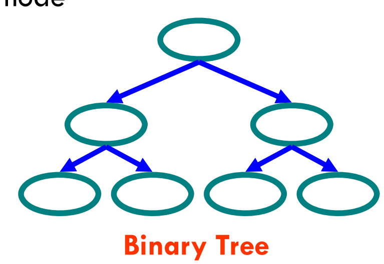
- Terminology
  - Root ⇒ no parent
  - Leaf ⇒ no child
  - Interior ⇒ non-leaf
  - Height ⇒ distance from root to leaf



Trees Data Structures

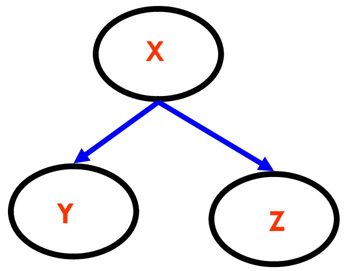
- Tree
  - Nodes
  - Each node can have 0 or more children
  - A node can have at most one parent
- Binary tree
  - Tree with 0–2 children per node





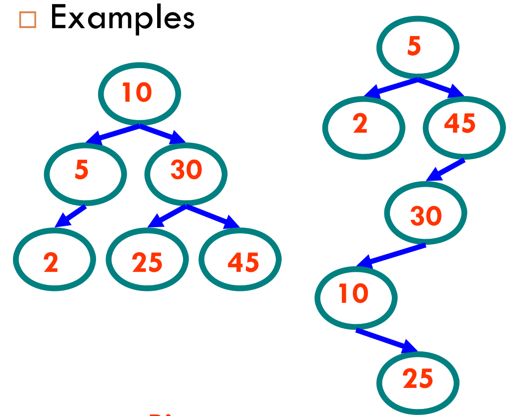
Binary Tree Constructions

- Constructing a tree is done by inserting each new node at a leaf position
- We assume the Left Child is ***less than*** the Right Child.
- Key property
  - Value at node
    - Smaller values in left subtree
    - Larger values in right subtree
  - Example
    - X > Y
    - X < Z

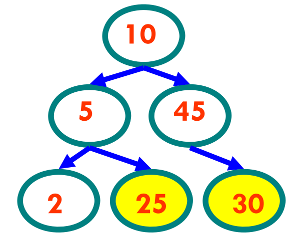


Binary Tree Constructions - Example

- Examples



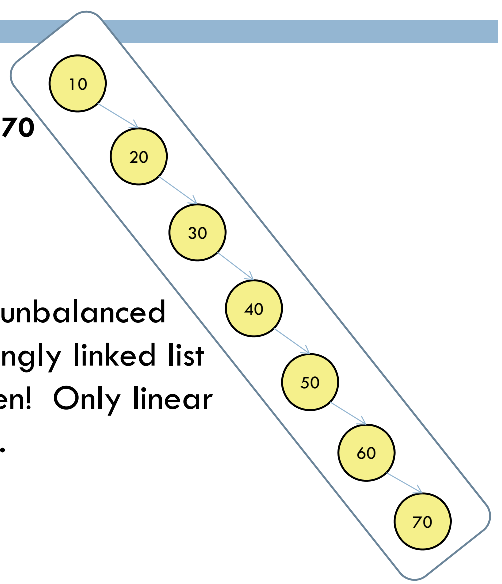
Binary trees



Not a binary tree

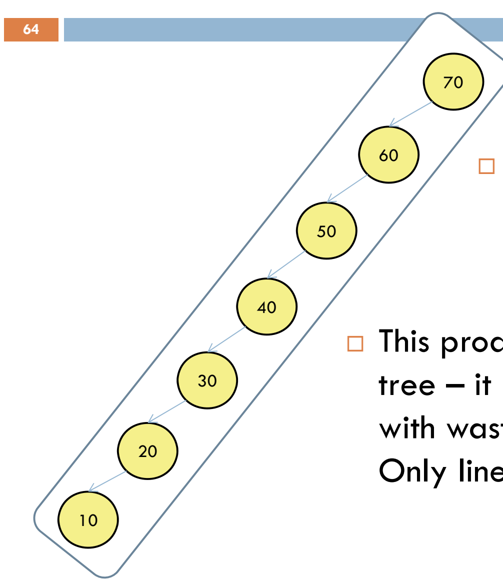
Constructing a Tree - Example 1

- Input:
  - **10, 20, 30, 40, 50, 60, 70**



- This produces a highly unbalanced tree – it is actually a singly linked list with wasted left children! Only linear search can be applied.

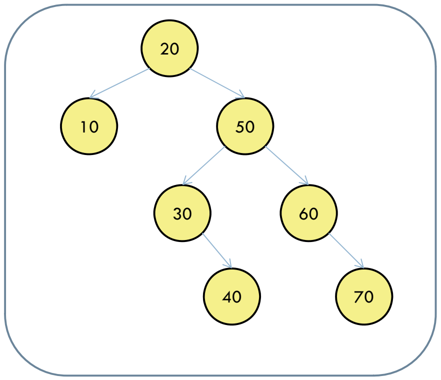
Constructing a Tree - Example 2



- Input:
  - **70, 60, 50, 40, 30, 20, 10**
- This produces a highly unbalanced tree – it is actually a singly linked list with wasted right children!
  Only linear search can be applied.

Constructing a Tree - Example 3

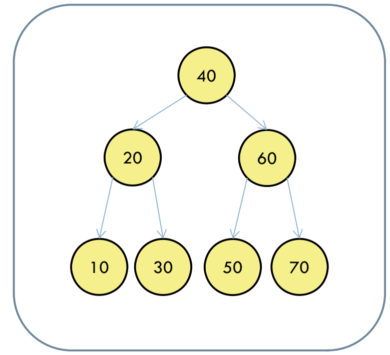
- Input:
  - **20, 50, 30, 40, 10, 60, 70**



  - This version is better balanced but search is between O(n) and O(log n)

Constructing a Tree - Example 4

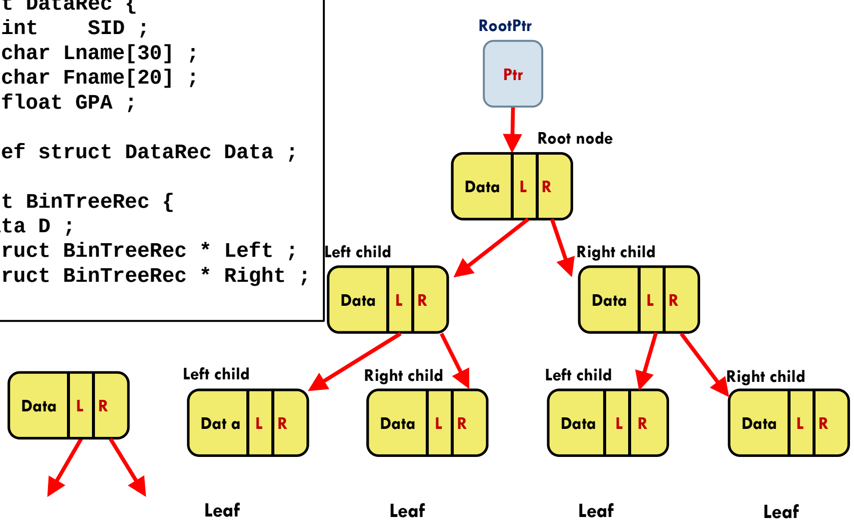
- Input
  - **40, 20, 10, 30, 60, 50, 70**



- This version is perfectly balanced and search is O(log n)

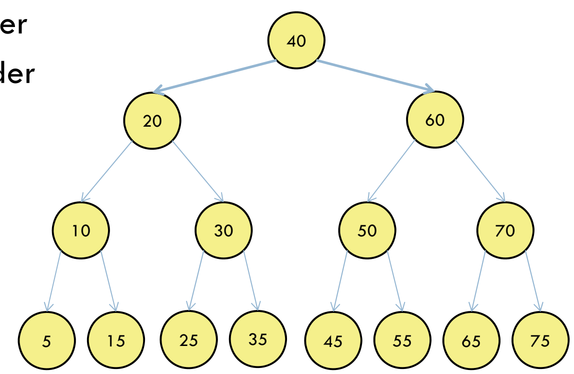
Binary Tree Definition


```
struct DataRec {
     int    SID ;
     char Lname[30] ;
     char Fname[20] ;
     float GPA ;
}
typedef struct DataRec Data ;

struct BinTreeRec {
   Data D ;
   struct BinTreeRec * Left ;
   struct BinTreeRec * Right ;
}
```



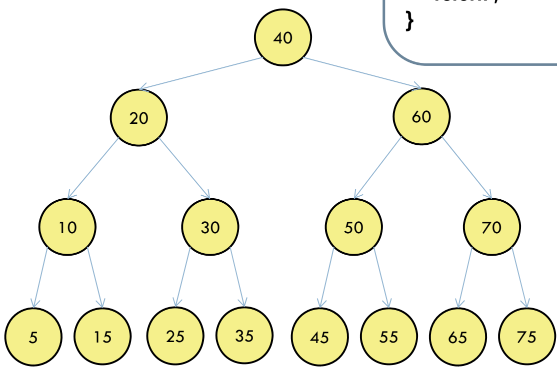
Tree Traversal Techniques

- Traversals
  - Inorder
  - Preorder
  - Postorder



Inorder

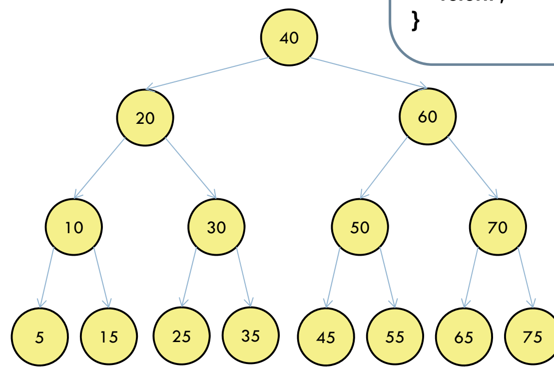

```
void inOrder ( TreeNodePtr_t  TPtr ) {
    if( TPtr != NULL ) {
        inOrder( TPtr->leftChild ) ;  \\ go left
        outputNode( TPtr ) ;          \\  output node
        inOrder( TPtr->rightChild ) ;  \\ go right
    }
    return ;
}
```




```
inOrder:    5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75
```

Preorder

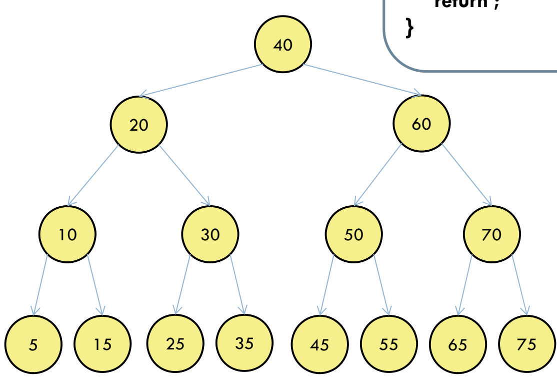

```
void preOrder ( TreeNodePtr_t  TPtr ) {
    if( TPtr != NULL ) {
        outputNode( TPtr ) ;          \\  output node
        preOrder( TPtr->leftChild ) ;  \\ go left
        preOrder( TPtr->rightChild ) ;  \\ go right
    }
    return ;
}
```




```
preOrder:  40, 20, 10, 5, 15, 30, 25, 35, 60, 50, 45, 55, 70, 65, 75
```

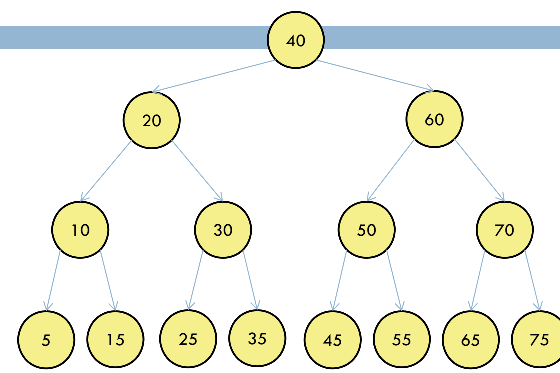
Postorder


```
void postOrder ( TreeNodePtr_t  TPtr ) {
    if( TPtr != NULL ) {
        postOrder( TPtr->leftChild ) ;  \\ go left
        postOrder( TPtr->rightChild ) ;  \\ go right
        outputNode( TPtr ) ;          \\  output node
    }
    return ;
}
```




```
postOrder:  5, 15, 10, 25, 35, 30, 20, 45, 55, 50, 65, 75, 70, 60, 40
```

Tree Traversal Techniques




```
inOrder:    5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75
preOrder:  40, 20, 10, 5, 15, 30, 25, 35, 60, 50, 45, 55, 70, 65, 75
postOrder:  5, 15, 10, 25, 35, 30, 20, 45, 55, 50, 65, 75, 70, 60, 40
```

Tree Data Structures

- There are several important algorithms necessary to work with tree data structures, including:
  - InsertTreeNode
  - DeleteTreeNode
  - RotateNodeLeft – used to reorder parent-child node relationships
  - RotateNodeRight
  - FindTreeHeight
- This subject is covered in much more detail in 2nd year courses on data structures and advanced programming

Lecture 9: Summary

- Linked List Variations and useful functions
  - Doubly Linked List
  - Stacks
  - Queues
  - trees
- Reading – Chapter 12: Data Structures
  - Abstract data structures, dynamic memory allocation, using pointers and self-referential data structures, linked lists.
- Assignment
  - Deadline of the fifth assignment is on **Wednesday, March 29**.
- Notes
  - There will be NO LAB during the last week of classes.
  - Any queries regarding the lab exercises:
    - Dr. Kent will be available on **Tuesday April 4 between 1-4pm** in Erie Hall teaching lab.
    - Osama will be available on **Wednesday 5th of April from 5:30 pm until 7:00 pm.**

Bonus Coupon

- You must present this coupon **in person and in class before the end of the last lecture to the instructor**.

Late Assignment Penalty Waiver/Bonus Coupon

Usage: This coupon is primarily intended to provide the student extra time to complete an assignment properly if the need arises for whatever reason. Use this coupon if you feel you are unable to submit the assignment on time. It grants you to extend the original deadline by 72 hours without being penalized for late submission. If you successfully complete ALL assignments on time, and do not use this coupon for the purpose of time extension, then you may submit this coupon in person to the instructor on or before the last official lecture and earn 1% toward your cumulative total of the assignments. For late waiver, the student must notify the instructor and submit the coupon in person within 24 hours from the original assignment due date. If you submit the coupon for late waiver and do not submit the assignment, or submit it after 72 hours from the original due date, then you will lose the coupon value and your assignment will not be graded.

Rules: Limit one per student. Not valid with any other offer. Coupon expires at **the end of your last regular lecture**. Non-negotiable, not redeemable for marks other than that specified above, not transferable. No late coupon submission is accepted once the offer expires. The instructor reserves the right to refuse or cancel this coupon offer at any time without notification. You must present this coupon **in person and in class before the end of the last lecture to the instructor**.

THIS COUPON MAY BE USED ONLY ONCE! PLEASE COMPLETE ALL FIELDS LEGIBLY.

PRINT NAME: ______________________________ LECTURE SECTION: __02___ LAB SECTION: _______

Last name, First name

STUDENT ID: ______________________________ DATE: ______________________

CHECK ONE: ❑ WAIVE LATE PENALTY FOR ASSIGNMENT NUMBER: ______ DUE: ______________

(This grants the student up to 72 hours to submit the assignment from the original due date/time without penalty. Submitting the assignment after the 72 hours grace period is considered late and will be penalized accordingly; not submitting the assignment void the coupon and as a result the coupon can no longer be used for "bonus" and the assignment mark will be zero. This coupon does not apply for labs.)

or:

❑ CLAIM 1 BONUS POINT **(Valid only if ALL assignments were successfully submitted on time, regardless of the mark obtained.)**