Data Structure CS301

Lecture No. 01

Introduction to Data Structures: Data structures help us to organize the data in the computer, resulting in more efficient programs. An efficient program executes faster and helps minimize the usage of resources like memory, disk.

What does organizing the data mean?

It means that the data should be arranged in a way that it is easily accessible.

Data Structure Philosophy

a data structure may be better than another one in all situations.
There are three basic things associated with data structures. A data structure requires:

1. space for each data item it stores
2. time to perform each basic operation
3. programming effort

What is *x*? *x* is a name of collection of items. int x[6];
'*x*' is an array's name but there is no variable *x*. '*x*' is not an *lvalue*. If some variable can be written on the left-hand side of an assignment statement, this is *lvalue* variable.

List data structure

The *List* data structure is among the most generic of data structures. A list is the collection of items of the same type. The items, or elements of the list, are stored in some particular order. List is a set of elements in a linear order.

Operation Name	Description
createList()	Create a new list (presumably empty)
copy()	Set one list to be a copy of another
clear();	Clear a list (remove all elements)
insert(X, ?)	Insert element X at a particular position in the list
remove(?)	Remove element at some position in the list
get(?)	Get element at a given position
update(X, ?)	Replace the element at a given position with X
find(X)	Determine if the element X is in the list
length()	Returns the length of the list.

If we use the "current" marker, the following four methods would be useful:

Functions	Description
start()	Moves the "current" pointer to the very first element
tail()	Moves the "current" pointer to the very last element
next()	Move the current position forward one element
back()	Move the current position backward one element

Lecture No. 02

How to implement the interface. Suppose we want to create a list of integers. For this purpose, the methods of the list can be implemented with the use of an array inside.
for simplification purposes, it is good to use the index from 1.

Add Method

To add the new element (9) to the list at the current position, at first, we have to make space for this element. For this purpose, we shift every element on the right of 8 (the current position) to one place on the right. (2, 6, 8, 9, 7, 1)
Now in the second step, we put the element 9 at the empty space i.e. position 4.

next Method

We have talked that the next method moves the current position one position forward. In this method, we do not add a new element to the list but simply move the pointer one element ahead.

remove Method

We have seen that the *add* method adds an element in the list. The *remove* method removes the element residing at the current position.

find Method

The *find*
(x) function is used to find a specific element in the array. We pass the element, which is to be found, as an argument to the *find* function. This function then traverses the array until the specific element is found. If the element is found, this

function sets the current position to it and returns 1 i.e. true. On the other hand, if the element is not found, the function returns 0 i.e. false. This indicates that the element was not found. Following is the code of this *find(x)* function in C++.

```
int find (int x)
{
        int j ;
        for (j = 1; j < size + 1; j++ )
                if (A[j] == x )
                        break ;
        if ( j < size + 1) // x is found
{
        current = j ; //current points to the position where x found
        return 1 ; // return true
}
        return 0 ; //return false, x is not found
}
```

get() method , used to get the element from the current position in the array.
method *length( )*.This method returns the size of the list.
back() method decreases the value of variable *current* by 1. In other words, it moves the current position one element backward. current -- ;
The *start()* method sets the current position to the first element of the list.

Analysis of Array List

Time is the major factor to see the efficiency of a program.

List using Linked Memory

In an array, the memory cells of the array are linked with each other. It means that the memory of the array is contiguous. In an array, it is impossible that one element of the array is located at a memory location while the other element is located somewhere far from it in the memory. It is located in very next location in the memory.
Linked List
For the utilization of the concept of linked memory, we usually define a structure, called linked list. To form a linked list, at first, we define a node. A node comprises two fields. i.e. the *object* field that holds the actual list element and the *next* that holds the starting location of the next node.
A chain of these nodes forms a linked list.
We use *head* and *current* variable names instead of using the memory address in numbers for starting and the current nodes.

Lecture No. 03

There is the limitation that array being of fixed size can only store a fixed number of elements. Therefore, no more elements can be stored after the size of the array is reached.
In order to resolve this, we adopted a new data structure called *linked list.*

Linked List Operations

The linked list data structure provides operations to work on the nodes inside the list. The *Add(9)* is used to create a new node in the memory at the current position to hold '*9*'. You must remember while working with arrays, to add an element at the current position that all the elements after the current position were
shifted to the right and then the element was added to the empty slot.(***Node * newNode = new Node(9);)***
Hence, the whole statement means:"Call the constructor of the *Node* class and pass it '*9*' as a parameter. After constructing the object in memory, give me the starting memory address of the object.That address will be stored in the pointer variable *newNode*."
When we write class in C++, normally, we make two files (*.h* and *.cpp*) for a class. The *.h* file contains the declarations of *public* and *private* members of that class. The *public* methods are essentially the interface of the class to be employed by the users of this class. The *.cpp* file contains the implementation for the class methods that has the actual code. But this is not mandatory. In the code given above, we have only one file *.cpp,* instead of separating into two files.
The second method in the above-mentioned class is *set() that* accepts a parameter of type *int* while returning back nothing. The accepted parameter is assigned to the internal data member *object*.
The next method is *getNext()* *which* returns a pointer to an object of type *Node* lying somewhere in the memory. It returns *nextNode i.e.* a pointer to an object of type *Node*. As discussed above, *nextNode* contains the address of next node in the linked list.

The last method of the class is *setNext() that* accepts a pointer of type *Node*, further assigned to *nextNode* data member of the object. This method is used to connect the next node of the linked list with the current object. It is passed an address of the next node in the linked list.
(We normally use the arrow (-> i.e. minus sign and then the greater than sign) to manipulate the structure's and Class's data with pointers. So to access the name with *sptr* we will write: sptr->name; While accessing through a simple variable, use dot operator i.e. s1.name While accessing through the pointer to structure, use arrow operator i.e. sptr- >name;)
Lecture No. 04

Methods of Linked List

the *start* method that has the following code.

```
// position currentNode and lastCurrentNode at first element
void start() {
lastCurrentNode = headNode;
currentNode = headNode;
};
```

We will now see how a node can be removed from the link list. We use the method *remove* for this purpose.

```
void remove() {
            if( currentNode != NULL && currentNode != headNode) {
(step 1)                lastCurrentNode->setNext(currentNode->getNext());
(step 2)                delete currentNode;
(step 3)                currentNode = lastCurrentNode->getNext();
(step 4)                size--;
}
};
```

The next method is *length()* that simply returns the size of the list. The code is as follows:

```
// returns the size of the list
            int length()
            {
            return size;
            };
```

Analysis of Link List

add
For the addition purposes, we simply insert the new node after the current node. So 'add' is a one-step operation. We insert a new node after the current node in the chain.
if we have to add an element in the centre of the array, the space for it is created at first. For this, all the elements that are after the current pointer in the array, should be shifted one place to the right.
remove
Remove is also a one-step operation. The node before and after the node to be removed is connected to each other. Update the current pointer. Then the node to be removed is deleted. As a result, the node to be removed is deleted.
find
The worst-case in find is that we may have to search the entire list. In find, we have to search some particular element say *x*. If found, the *currentNode* pointer is moved at that node. As there is no order in the list, we have to start search from the beginning of the list. We have to check the value of each node and compare it with x (value to be searched). If found, it returns true and points the *currentNode* pointer at that node otherwise return false. Suppose that *x* is not in the list, in this case, we have to search the list from start to end and return false.
back
In the back method, we move the *current* pointer one position back. Moving the *current* pointer back, one requires traversing the list from the start until the node whose *next* pointer points to current node. Our link list is singly linked list i.e. we can move in one direction from start towards end.

Doubly-linked List

In doubly-link list, a programmer uses two pointers in the node, i.e. one to point to next node and the other to point to the previous node. Now our node factory will create a node with three parts. First part is *prev* i.e. the pointer pointing to the previous node, second part is *element*, containing the data to be inserted in the list.

Circularly-linked lists

The next field in the last node in a singly-linked list is set to NULL. The same is the case in the doubly-linked list. Moving along a singly-linked list has to be done in a watchful manner. Doubly-linked lists have two NULL pointers i.e.

prev in the first node and *next* in the last node. A way around this potential hazard is to link the last node with the first node in the list to create a *circularly-linked list*. We have connected the last node with the first node. It means that the *next* of the last node is pointing towards the first node.

Lecture No. 05

Benefits of using circular list

While solving the Josephus problem, it was witnessed that the usage of circular linked list helped us make the solution trivial. We had to just write a code of some lines that solved the whole problem.

Abstract Data Type

A data type is a collection of values and a set of operations on those values. That collection and these operations form a mathematical construct that may be implemented with the use of a particular hardware or software data structure. The term abstract data type (ADT) refers to the basic mathematical concept that defines the data type.

Stacks

Let's talk about another important data structure. You must have a fair idea of stacks. Some examples of stacks in real life are stack of books, stack of plates etc. We can add new items at the top of the stack or remove them from the top. We can only access the elements of the stack at the top. Following is the definition of stacks. "Stack is a collection of elements arranged in a linear order"

Let's talk about the interface methods of the stacks. Some important methods are:

Method Name Description

Method Name	**Description**
push(x)	Insert x as the top element of the stack
pop()	Remove the top element of the stack and return it.
top()	Return the top element without removing it from the stack.

The *push(x)* method will take an element and insert it at the top of the stack. This element will become top element. The *pop()* method will remove the top element of the stock and return it to the calling program. The *top()* method returns the top-most stack element but does not remove it from the stack.

The last element to go into the stack is the first to come out. That is why, a stack is known as *LIFO* (Last In First Out) structure.

The code of *pop()* method is:

```
int pop()
{
        return A[curren t--];
}
```

The code of *push* method is:

```
void push(int x)
{

int isEmpty()
{
        return ( current == -1 );
}
```

```
int isFull()
{
return ( current == size-1);
}

        A[++current] = x;
}
```

The code of the *top()* method is:

```
int top()
{
        return A[current];
}
```

Lecture No. 06

Stack Using Linked List

We can avoid the size limitation of a stack implemented with an array, with the help of a linked list to hold the stack elements.

stack Implementation: Array or Linked List

- since are the possible reasons to prefer one implementation to the other. Allocating and de-allocating m allocated array. what are the possible reasons to prefer one implementation to the other? Allocating and de-allocating m allocated array.

- List uses as much memory as required by the nodes. In contrast, array requires allocation ahead of time.

- List pointers (*head, next*) require extra memory. Consider the manipulation of array elements. We can set and get the individual elements with the use of the array index; we don't need to have additional elements or pointers to access them. But in case of linked list, within each node of the list, we have one pointer element called *next*, pointing to the next node of the list. Therefore, for 1000 nodes stack implemented using list, there will be 1000 extra pointer variables. Remember that stack is implemented using 'singly-linked' list. Otherwise, for doubly linked list, this overhead is also doubled as two pointer variables are stored within each node in that case.

Array has an upper limit whereas list is limited by dynamic memory allocation.
We could write +*AB*, the operator is written before the operands *A* and *B*. These kinds of expressions are called *Prefix* Expressions. We can also write it as *AB*+, the operator i s written after the operands *A* and *B*.
This expression is called *Postfix* expression.

Precedence of Operators

There are five binary operators, called *addition*, *subtraction*, *multiplication*, *division* and *exponentiation*. We are aware of some other binary operators.

Exponentiation □	Multiplication/division *, /	Addition/subtraction +, -

Lecture No. 07

Evaluating postfix expressions

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

The last expression seems a bit confusing but may prove simple by following the rules in letter and spirit. In the postfix form, parentheses are not used. Consider the infix expressions as '4+3*5' and '(4+3)*5'. The parentheses are not needed in the first but are necessary in the second expression. The postfix forms are:

4+3*5 435*+ (4+3)*5 43+5*

```
Stack s;                    // declare a stack
while( not end of input ) { // not end of postfix expression
        e = get next element of input
        if( e is an operand )
                s.push( e );
        else {
        op2 = s.pop();
        op1 = s.pop();
        value = result of applying operator 'e' to op1 and op2;
        s.push( value );
        }
}
finalresult = s.pop();
```

Lecture No. 08

C++ Templates

We can use C++ templates for stack and other data structures. We have seen that stack is used to store the operands while evaluating an expression. These operands may be integers, floating points and even variables. We push and pop the operands to and from the stack. In the conversion of an expression, a programmer uses the stack for storing the operators like +, *, -, and / etc

In C++ programming, we will have to create two classes *FloatStack* and *CharStack* for operands and operators respectively. These classes of stack have the same implementation.

In C++ language, a template is a function or class that is written with a generic data type. When a programmer uses this function or class, the generic data type is replaced with the data type, needed to be used in the template function or in the template class. We only give the data type or our choice with calling a template function or creating an object of the template class. The compiler automatically creates a version of that function or class with that specified data type.

The templates are so important that C++ provides a library in which a large number of common use functions are provided as templates. This library is a part of the official standard of C++. It is called STL i.e. Standard Template Library.

Function Call Stack: Whenever a programmer calls a function, he or she passes some arguments or parameters to the function. The function does work on these arguments and return a value to the calling function or program. This value is known as the return value of the function. We declare some variables inside the function which are local variables of the function. These variables are demolished when the execution of the function ends. If there are variables in the function that need to be preserved, we have to take care of them. For this purpose, we use global variables or return a pointer to that variable.

Lecture No. 09

Stack is used in function calling while heap area is utilized at the time of memory allocation in dynamic manner.

Parameters (F)	Parameters (F)	Parameters (F)

Local variables (F) Return address (F)	Local variables (F) Return address (F)	Local variables (F) Return address (F)
Parameters (G)	Parameters (G) Local variables (G) Return address (G)	

The above diagrams depict the layout of the stack when a function *F* calls a function *G*. Here **sp** stands for stack pointer. At the very left, you will find the layout of the stack just before function *F* calls function *G*. The parameters passed to function *F* are firstly inserted inside the stack. These are followed by the local variables of the function *F and finally* the memory address to return back after the function *F* finishes.
Just before function is made to the function *G*, the parameters being passed to the function *G*, are inserted into the stack. In the next diagram, there is layout of the stack on the right side after the call to the function *G*. Clearly, the local variables of the function G are inserted into the stack after its parameters and the return address. If there are no local variables for a function, the return address is inserted (pushed) on to the stack. The layout of the stack, when the function *G* finishes execution is shown on the right. You can see that the local variables of function G are no more in the stack. They have been removed permanently along with the parameters passed to the function G. Now, it is clear that when a function call is made, all local variables of the called function and the parameters passed to it, are pushed on to the stack and are destroyed, soon after the the completion of the called function's execution. In C/C++ language, the variables declared as *static* are not pushed on the stack. Rather, these are stored in another separate section allocated for *static* data of a program. It is not destroyed till the end of the process's execution.
Queues: A queue is a linear data structure into which items can only be inserted at one end and removed from the other. In contrast to the stack, which is a LIFO (Last In First Out) structure, a queue is a FIFO (First In First Out) structure. The usage of queue in daily life is pretty common. For example, we queue up while depositing a utility bill or purchasing a ticket. The objective of that queue is to serve persons in their arrival order; the first coming person is served first.
Queue Operations: The queue data structure supports the following operations:

Operation	**Description**
enqueue(X)	Place X at the *rear* of the queue.
dequeue()	Remove the *front* element and return it.
front()	Return *front* element without removing it.
isEmpty()	Return TRUE if queue is empty, FALSE otherwise

Implementing Queue: There are certain points related to the implementation of the queue. Suppose we are implementing queue with the help of the linked -list structure. Following are the key points associated with the linked list implementations:
- Insert works in constant time for either end of a linked list.
- Remove works in constant time only.
- Seems best that head of the linked list be the front of the queue so that all removes will be from the front.
- Inserts will be at the end of the list.

```
        /* Remove element from the front */
int dequeue()
{
int x = front->get();
Node* p = front;
front = front->getNext();
delete p;
return x;
}
/* Insert an element in the rear */
void enqueue(int x)
{
Node* newNode = new Node();
newNode->set(x);
newNode->setNext(NULL);
rear->setNext(newNode);
rear = newNode;
}
```

Queue using Array

A programmer keeps few important considerations into view account before implementing a queue with the help of an array: If we use an array to hold the queue elements, both insertions and removal at the front (start) of the array are expensive. This is due to the fact that we may have to shift up to “n” elements.
For the stack, we needed only one end but for a queue, both are required. To get around this, we will not shift upon removal of an element.

Although, we have insert and removal operations running in constantly, yet we created a new problem that we cannot insert new elements even though there are two places available at the start of the array. The solution to this problem lies in allowing the queue to *wrap around*. How can we *wrap around*? We can use circular array to implement the queue. We know how to make a linked list circular using pointers. Now we will see how can we make a circular array.

we will have to maintain four variables. *front* has the same index *2 while the*, *size* is *8. ‘ rear’* has moved to index *0* and *noElements* is *7*. Now, we can see that *rear* index has decreased instread of increasing. It has moved from index *7* to *0*. *front* is containing index *2 i.e.* higher than the index in *rear*. Let’ see, how do we implement the *enqueue()* method.

```
void enqueue( int x)
{
1. rear = (rear + 1) % size;
2. array[rear] = x;
3. noElements = noElements + 1;
}
```

the queue, rather the array has become full. It is important to understand, that queue does not have such characteristic to become full. Only its implementation array has become full. To resolve this problem, we can use linked list to implement a queue. For the moment, while working with array, we will write the method *isFull()*, to determine the fullness of the array.

```
int isFull()
{
return noElements == size;
}
int isEmpty()
{
return noElements == 0;
}
```

Use of Queues: We saw the uses of stack structure in *infix*, *prefix* and *postfix* expressions. Let’s see the usage of queue now. Out of the numerous uses of the queues, one of the most useful is *simulation*. A simulation program attempts to model a real-world phenomenon. Many popular video games are simulations, e.g., SimCity, Flight Simulator etc. Each object and action in the simulation has a counterpart in the real world. Computer simulation is very powerful tool and it is used in different high tech industries, especially in engineering projects. For example, it is used in aero plane manufacturing. Actually Computer
Simulation is full-fledged subject of Computer Science and contains very complex Mathematics, sometimes. For example, simulation of computer networks, traffic networks etc.

Lecture No. 10

Simulation Models: Let’s discuss little bit about the simulation models. Two common models of simulation are time-based simulation and event-based simulation. In time-based simulation, we maintain a timeline or a clock. The clock ticks and things happen when the time reaches the moment of an event.
Suppose we have a clock in the computer. The minute hand moves after every minute. We know the time of the customer’s entry into the bank and are aware that his transaction takes 5 minutes. The clock is ticking and after 5 minutes, we will ask the customer to leave the bank. In the program, we will represent the person with some object. As the clock continues ticking, we will treat all the customers in this way.
Note that when the customer goes to some teller, he will take 5 minutes for his transaction. During this time, the clock keeps on ticking. The program will do nothing during this time period. Although some other customer can enter the bank. In this model, the clock will be ticking during the transaction time and no other activity will take place during this time. If the program is in some loop, it will do nothing in that loop until the completion of the transaction time.

Priority Queue: As stated earlier, the queue is a FIFO (First in first out) structure. In daily life, you have also seen that it is not true that a person, who comes first, leaves first from the queue. Let’s take the example of traffic. Traffic is stopped at the signal. The vehicles are in a queue. When the signal turns green, vehicles starts moving. The vehicles which are at the front of the queue will cross the crossing first. Suppose an ambulance comes from behind. Here ambulance should be given priority. It will bypass the queue and cross the intersection. Sometimes, we have queues that are not FIFO i.e. the person who comes first may not leave first. We can develop such queues in which the condition for leaving the queue is not to enter first. There may be some priority. Here we will also see the events of future like the customer is coming at what time and leaving at what time. We will arrange all these events and insert them in a priority queue. We will develop the queue in such a way that we will get the event which is going to happen first of all in the future. This data structure is known as priority queue. In a sense, FIFO is a special case of priority queue in which priority is given to the time of arrival. That means the person who comes first has the higher priority while the one who comes later, has the low priority. You will see the priority queue being used at many places especially in the operating systems. In operating systems, we have queue of different processes. If some process comes with higher priority, it will be processed first. Here we have seen a variation of queue. We will use the priority queue in the simulation. The events will be inserted in the queue and the event going to occur first in future, will be popped.

Lecture No. 11

Implementation of Priority Queue: in the priority queue, we put the elements in the queue to get them from the queue with a priority of the elements. Following is the C++ code of the priority queue.

Tree:

Now let's talk about a data structure called tree. This is an important data structure. This data structure is used in many algorithms. We will use it in most of our assignments. The data structures that we have discussed in previous lectures are linear data structures. The linked list and stack are linear data structures. In these structures, the elements are in a line. We put and get elements in and from a stack in linear order. Queue is also a linear data structure as a line is developed in it. There are a number of applications where linear data structures are not appropriate. In such cases, there is need of some non-linear data structure. Some examples will show us that why nonlinear data structures are important. Tree is one of the non-linear data structures.

There may be situations where the data, in our programs or applications, is not in the linear order. There is a relationship between the data that cannot be captured by a nked list or other linear data structure. Here we need a data structure like tree.

Binary Tree: The mathematical definition of a binary tree is

"A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary trees called the left and right sub-trees".

Each element of a binary tree is called a node of the tree.

Terminologies of a binary tree: Now let's discuss different terminologies of the binary tree. We will use these terminologies in our different algorithms.

Parent Left descendant Right descendant Leaf nodes

Strictly Binary Tree: There is a version of the binary tree, called strictly binary tree. A binary tree is said to be a strictly binary tree if every non-leaf node in a binary tree has non-empty left and right subtrees.

Level: The level of a node in a binary tree is defined as follows:

- Root has level 0,
- Level of any other node is one more than the level its parent (father).
- the *depth* of a binary tree is the maximum level of any leaf in the tree.

Complete Binary Tree: the definition of the complete binary tree is

"A complete binary tree of depth d is the strictly binary tree all of whose leaves are at level d".

Operations on Binary Tree: We can define different operations on binary trees. If *p* is pointing to a node in an existing tree, then

- left(*p*) returns pointer to the left subtree
- right(*p*) returns pointer to right subtree
- parent(*p*) returns the father of *p*
- brother(*p*) returns brother of *p*.
- info(p) returns content of the node.

Tips

- A priority queue is a variation of queue that does not follow FIFO rule.
- Tree is a non-linear data structure.
- The maximum level of any leaf in a binary tree is called the depth of the tree.
- Other than the root node, the level of each node is one more than the level of its parent node.
- A complete binary tree is necessarily a strictly binary tree but not vice versa.
- At any level *k*, there are 2*k* nodes at that level in a complete binary tree.
- The total number of nodes in a complete binary tree of depth *d* is 2*d+1* – 1.
- In a complete binary tree there are 2*d* leaf nodes and 2*d* – 1 non-leaf nodes.

Lecture No. 12

Applications of Binary Tree: Binary tree is useful structure when two-way decisions are made at each point. Suppose we want to find all duplicates in a list of the following numbers:
14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5
This list may comprise numbers of any nature. For example, roll numbers, telephone numbers or voter's list. In addition to the presence of duplicate number, we may also require the frequency of numbers in the list. As it is a small list, so only a cursory view may reveal that there are some duplicate numbers present in this list. Practically, this list can be of very huge size ranging to thousands or millions.

So, the solution lies in reducing the number of comparisons. The number of comparisons can be drastically reduced with the help of a binary tree. The benefits of linked list are there, also the tree grows dynamically like the linked list. The binary tree is built in a special way. The first number in the list is placed in a node, designated as the *root* of the binary tree. Initially, both left and right sub-trees of the *root* are empty. We take the next number and compare it with the number placed in the *root*. If it is the same, this means the presence of a duplicate. Otherwise, we create a new tree node and put the new number in it. The new node is turned into the left child of the *root* node if the second number is less than the one in the *root*. The new node is turned into the right child if the number is greater than the one in the root.

It is pertinent to note that this is a binary tree with two sub-nodes or children of each node. We have not seen the advantage of binary tree, the one we were earlier talking about i.e. it will reduce the number of comparisons. Previously, we found that search operation becomes troublesome and slower as the size of list grow. We will see the benefit of using binary tree over linked list later. Firstly, we will see how the tree is implemented.

Lecture No. 13

Cost of Search

Consider the previous example where we inserted the number 17 in the tree. We executed a *while* loop in the *insert* method and carried out a comparison in *while* loop. If the comparison is true, it will reflect that in this case, the number in the node where the pointer *p* is pointing is not equal to 17 and also *q* is not NULL. Then we move *p* actually *q* to the left or right side. This means that if the condition of the *while* loop is true then we go one level down in the tree. Thus we can understand it easily that if there is a tree of 6 levels, the while loop will execute maximum 6 times. We conclude from it that in a given binary tree of depth *d*, the maximum number of executions of the *while* loop will be equal to *d*. The code after the *while* loop will do the process depending upon the result of the *while* loop. It will insert the new number or display a message if the number was already there in the tree.

Suppose we have a complete binary tree in which there are 1000,000 nodes, then its depth *d* will be calculated in the following fashion.

$$d = \log2\ (1000000 + 1) – 1 = \log2\ (1000001) – 1= 20$$

In a tree, the search is very fast as compared to the linked list. If the tree is complete binary or near-to-complete, searching through 1000,000 numbers will require a maximum of 20 comparisons or in general, approximately *log*2*(n)*. Whereas in a linked list, the comparisons required could be a maximum of *n*.

Binary Search Tree

While discussing the search procedure, the tree for search was built in a specific order. The order was such that on the addition of a number in the tree, we compare it with a node. If it is less than this, it can be added to the left sub-tree of the node. Otherwise, it will be added on the right sub-tree. This way, the tree built by us has numbers less than the root in the left sub-tree and the numbers greater than the root in the right sub-tree. A binary tree with such a property that items in the left sub-tree are smaller than the root and items in the right sub-tree are larger than the root is called a *binary search tree* (BST). The searching and sorting operations are very common in computer science. We will be discussing them many times during this course. In most of the cases, we sort the data before a search operation. The building process of a binary search tree is actually a process of storing the data in a sorted form. The BST has many variations, which will be discussed later. The BST and its variations play an important role in searching algorithms. As data in a BST is in an order, it may also be termed as ordered tree.

Traversing a Binary Tree

The code of the *preorder* method.

```
void preorder(TreeNode<int>* treeNode)
{
        if( treeNode != NULL )
{
        cout << *(treeNode->getInfo())<<" ";
}
        preorder(treeNode->getLeft());
        preorder(treeNode->getRight());
}
```

Here is the code of the *inorder* function.

```
void inorder(TreeNode<int>* treeNode)
{
        if( treeNode != NULL )
{
        inorder(treeNode->getLeft());
        cout << *(treeNode->getInfo())<<" ";
        inorder(treeNode->getRight());
}}
```

postorder method.

```
void postorder(TreeNode<int>* treeNode)
{
        if( treeNode != NULL )
{
        postorder(treeNode->getLeft());
        postorder(treeNode->getRight());
        cout << *(treeNode->getInfo())<<" ";
}}
```

Lecture No. 14

Recursive Calls

We know that function calls are made with the help of stacks. When a function calls some other function, the parameters and return address of the function is put in a stack. The local variables of the function are also located at the stack and the control is passed to the called function. When the called function starts execution, it performs its job by using these parameters and local variables. When there in the function there comes a return statement or when the function ends then the return address, already stored in the stack, is used and control goes back to the next statement after the calling function. Similarly when a function is calling to itself, there is no problem regarding the stack. We know, in recursion a function calls to itself instead of calling some other function. Thus the recursion is implemented in the way as other function calls are implemented.

The tree data structure by nature is a recursive data structure. In the coming lecture, we will see that most of the methods we write for tree operations are recursive. From the programming point of view, the recursion is implemented internally by using call stack. This stack is maintained by the run time environment. The recursion is there in the machine code generated by the compiler. It is the programmer's responsibility to provide a terminating condition to stop the recursion. Otherwise, it will become an infinite recursion. If we do not put a terminating condition, the recursive calls will be continued and the call stack will go on increasing. We know that when a program executes, it becomes a process and some memory is allocated to it by the system for its call stack. This memory has a limit. Thus when the recursive calls do not end, there is no memory left to be used for increasing call stack after a certain time. This will crash the program or the program will halt. So we should be very careful while using the recursive calls and ensure the provision of a terminating condition in the recursive calls.

This is also an important aspect of programming. Program readability is also an issue. Suppose you have written some program. Will you understand if after some months that why you have written this and how? The first thing may be that what you are doing in the program and how you do it? After going through the program, you will remember it and recall that you have used this data structure for some purpose. Always comment the code. Comment the data structure, logic and algorithm of the program. Recursive procedure is an elegant procedure. Both the data structure and procedure are recursive. We have traversed a tree with only three four statements. No matter whatever is the size of the tree?

When the recursion happens with the help of function calls and stack.. There are some other values also included. It has return address, local variables and parameters. When a function calls another function irrespective of recursive or non-recursive like function F is calling function G. it will take time to put the values in the stack. If you create your own stack, it takes time for creation and then push and pop will also consume time. Now think that which one of these will take more time. The situation is this that function calls always takes place using stack irrespective of the language. The implementation of using stack is implemented very efficiently in the Assembly language. In the computer architecture or Assembly language program, you will study that the manipulation of stack calls that is push and pop and other methods are very efficiently coded. Now you may think that there is a lot of work needed for the recursive calls and non- recursive calls are faster. If you think that the non-recursive function will work fast, it is wrong. The experience shows that if recursive calls are applied on recursive data structures, it will be more efficient in comparison with the non-recursive calls written by you. We will see more data structures which are inherently recursive. If the data structures are recursive, then try to write recursive methods. We will further discuss binary tree, binary search tree and see more examples. With the use of recursion, our program will be small, efficient and less error prone. While doing programming, we should avoid errors. We don't want to see there are errors while writing a program and executing it.

Lecture No. 15

Level-order Traversal of a Binary Tree: its implementation is simple using non-recursive method and by employing queu*e* instead of stack. A queue is a FIFO structure, which can make the level-order traversal easier

Storing Other Types of Data in Binary Tree: Until now, we have used to place *int* numbers in the tree nodes. We were using *int* numbers because they were easier to understand for sorting or comparison problems. We can put any data type in tree nodes depending upon the problem where tree is employed. For example, if we want to enter the names of the people in the telephone directory in the binary tree, we build a binary tree of strings.

Binary Search Tree (BST) with Strings:

```
void wordTree()
{
TreeNode<char> * root = new TreeNode<char>();
static char * word[] = "babble", "fable", "jacket",
"backup", "eagle","daily","gain","bandit","abandon",
"abash","accuse","economy","adhere","advise","cease",
"debunk","feeder","genius","fetch","chain", NULL};
root->setInfo( word[0] );
for(i=1; word[i]; i++);
insert(root, word[i] );
inorder( root );
cout << endl;   }
```

Deleting a Node from BST:

Until now, we have been discussing about adding data elements in a binary tree but we may also require to delete some data (nodes) from a binary tree. Consider the case where we used binary tree to implement the telephone directory, when a person leaves a city, its telephone number from the directory is deleted.

It is common with many data structures that the hardest operation is deletion. Once we have found the node to be deleted, we need to consider several possibilities. For case1, if the node is a leaf, it can be deleted quite easily.

Lecture No. 16

Deleting a node in BST: In the previous lecture, we talked about deleting a node. Now we will discuss the ways to delete nodes of a BST (*Binary Search Tree*). Suppose you want to delete one of the nodes of BST. There are three cases for deleting a node from the BST. Case I: The node to be deleted is the leaf node i.e. it has no right or left child. It is very simple to delete such node. We make the pointer in the parent node pointing to this node as NULL. If the memory for the node has been dynamically allocated, we will release it. Case II: The node to be deleted has either left child (subtree) or right child (subtree). Case III: The node to be deleted has both the left and right children (subtree). This is the most difficult case.

```
        /* This method is used to remove a node from
the BST */
        TreeNode<int>* remove(TreeNode<int>* tree,
int info){
        TreeNode<int>* t;
        int cmp = info - *(tree->getInfo());
        if( cmp < 0 ){              // node to delete is in
left subtree
        t = remove(tree ->getLeft(), info);
        tree->setLeft( t );}
        else if( cmp > 0 ){
        t = remove(tree->getRight(), info);
        tree->setRight( t );}
        //two children, replace with inorder successor
        else if(tree->getLeft() != NULL && tree-
>getRight() != NULL ){
        TreeNode<int>* minNode;
        MinNode = findMin(tree->getRight());
        tree->setInfo( minNode->getInfo() );
        t= remove (tree->getRight(), *(minNode-
>getInfo()));
        tree->setRight( t );}
        else {
        treeNode<int>* nodeToDelete = tree;
        if( tree->getLeft() == NULL ) //will handle 0
children
        tree = tree->getRight();
        else if( tree->getRight() == NULL )
        tree = tree->getLeft();
        else tree = NULL;
        delete nodeToDelete; // release the memory
        }
        return tree;        }

        /* This method is used to find the minimum
node in a tree */
        TreeNode<int>* findMin(TreeNode<int>* tree)
        if( tree == NULL)
        return NULL;
        if( tree->getLeft() == NULL )
        return tree; // this is it.
        return findMin( tree->getLeft() );
        }
```

Lecture No. 17

Reference Variables:

The symbol &, used for reference variable has a few different purposes with respect to its occurrence in the code. In C++ programming, we have seen that when the ampersand sign i.e. & appears in front of a variable name, it is the address operator. It returns the address of the variable in front of which it is written. Thus for example, if *x* is a variable name, then *&x ;* will return the address of the variable *x*. In general we can say that

&variablename ;

Will return the address of the variable. We know that an address can be stored in a pointer.

To further understand this concept, let's suppose that there are following lines in our code.

```
int x ;                                              int* ptr = &x;
```

The first line declares a variable *x* of type *int* while the second one declares a pointer to *int* and assigns it the address of *x*. This address of variable *x* is returned by the & sign written in front of the variable *x*. Thus a pointer variable is initialized by assigning it the address of a variable. This address of the variable is gotten by using the & sign with the variable name.

The other place where & sign can appear is the signature of the function where it appears after the type of the parameter. Consider the insert and remove methods from BinarySearchTree class that were declared as the following.

```
void insert( const EType& x );
void remove( const EType& x );
```

Notice that the & sign is after the type. Here we define the class and functions as templates. Whenever, we use these methods, the type EType will be replaced with a proper data type.

Suppose that we have designed the BinarySearchTree class to hold the integers only. This means that there are no templates and the class can be defined only for integers. So the insert and remove methods of this class will be as follows.

```
void insert( const int& x );
void remove( const int& x );
```

Here, in the function signature, & sign after the type of the parameter indicates that the parameter is a reference variable. We can use this & sign with any data type i.e. built- in or user defined, of an argument to show that this is a reference variable. This is the syntax for the reference variable.

```
// Function 3

int intMinus3( int& oldVal)
{
        oldVal = oldVal – 3;
        return oldVal;
}
```

The & sign after the type in the signature (declaration line) indicates that this argument is a reference variable. Notice that & sign is used only in the function declaration, leaving no need for its use in the body of the function. The statements in the body of the function decrease the vale of *oldVal* by 3 and return it to the calling function.

```
void caller()
{
    int retVal;
    int myInt = 31;
    retVal = intMinus2( &myInt );
    cout << myInt << retVal;
}
```

Following is the function calling statement `retVal = intMinus2( &myInt );`

The '&' sign before the name of the variable *myInt* means that the address of the variable is being passed. The called function will take it as a pointer. Remember that we have declared the argument of the function *intMinus2* as *int* oldVal* that means this function takes a pointer as an argument.

This phenomenon of call by reference is actually implemented by the compiler by using pointers. The obtaining of address and de-referencing will be done behind the scene. We have no concern to do this. For a programmer, it is simply a renaming abstraction, in which we rename the argument variable of the caller function and use it in the called function.

Lecture No. 18

What can we do, if we do not want the objects created in a function to be destroyed? The answer to this is dynamic memory allocation. All the variables or objects created in a function that we want to access later are created on memory heap (sometimes called free store) using the dynamic memory allocation functions or operators like *new*. Heap is an area in computer memory that is allocated dynamically. You should remember that all the objects created using *new* operator have to be explicitly destroyed using the *delete* operator.

The *const* Keyword

The *const* keyword is used for something to be constant. The actual meanings depends on where it occurs but it generally means something is to held constant. There can be constant functions, constant variables or parameters etc.

The references are pointers internally, actually they are constant pointers. You cannot perform any kind of arithmetic manipulation with references that you normally do with pointers. You must be remembering when we wrote header file for binary tree class, we had used *const* keyword many times. The *const* keyword is often used in function signatures. The function signature is also called the function prototype where we mention the function name, its parameters and return type etc.

Here are some common uses of *const* keyword.

1. The const keyword appears before a function parameter. E.g., in a chess program:

```
int movePiece(const Piece & currentPiece)
```

The function *movePiece()* above is passed one parameter, which is passed by reference. By writing *const*, we are saying that parameter must remain constant for the life of the function. If we try to change value, for example, the parameter appears on the left side of the assignment, the compiler will generate an error. This also means that if the parameter is passed to another function, that function must not change it either.

Use of *const* with reference parameters is very common. This is puzzling; why are we passing something by reference and then make it constant, i.e., don't change it? Doesn't passing by reference mean we want to change it?

- The arithmetic operations we perform on pointers, cannot be performed on references
- Reference variables must be declared and initialized in one statement.
- To avoid dangling reference, don't return the reference of a local variable (transient) from a function.
- In functions that return reference, return global, static or dynamically allocated variables.
- The reference data types are used as ordinary variables without any dereference operator. We normally use arrow operator (->) with pointers.
- *const* objects cannot be assigned any other value.
- If an object is declard as *const* in a function then any further functions called from this function cannot change the value of the *const* object.

Lecture No. 19

We send a parameter to a function by using call by reference and put *const*with it. With the help of the reference variable, a function can change the value of the variable. But at the same time, we have used the *const*keyword so that it does not effect this change. With the reference parameter, we need not to make the copy of the object to send it to the calling function. In case of call by value, a copy of object is made and placed at the time of function calling in the activation record. Here the copy constructor is used to make a copy of the object. If we don't want the function to change the parameter without going for the use of time, memory creating and storing an entire copy of, it is advisable to use the reference parameter as *const*. By using the references, we are not making the copy. Moreover, with the *const* keyword, the function cannot change the object. The calling function has read only access to this object. It can use this object in the computation but can not change it. As we have marked it as constant, the function cannot alter it, even by mistake. The language is supportive in averting the mistakes.

There is another use of keyword *const*. The *const* keyword appears at the end of class member's function signature as:

```
EType& findMin( ) const;
```

This method is used to find the minimum data value in the binary tree. As you have noted in the method signature, we had written *const* at the end. Such a function cannot change or write to member variables of that class. Member variables are those which appear in the *public* or *private* part of the class. For example in the *BinaryTree*, we have *root* as a member variable. Also the *item* variable in the node class is the member variable. These are also called state variables of the class. When we create an object from the factory, it has these member variables and the methods of this class which manipulate the member variables. You will also use *set* and *get* methods, generally employed to set and get the values of the member variables. The member function can access and change the *public* and *private* member variables of a class. Suppose, we want that a member function can access the member variable but cannot change it. It means that we want to make the variables read only for that member function. To impose that constraint on the member function, a programmer can put the keyword *const* in the end of the function. This is the way in the C++ language. In other languages, there may be alternative methods to carry out it. These features are also available in other object oriented languages. This type of usage often appears in functions that are supposed to read and return member variables.

There is another use of *const*. The *const* keyword appears at the beginning of the return type in function signature:

```
const EType& findMin( ) const;
```

The return type of the *findMin()* function is *ETyper&* that means a reference is returned. At the start of the return type, we have *const*keyword. How is this implemented internally? There are two ways to achieve this. Firstly, the function puts the value in a register that is taken by the caller. Secondly, the function puts the value in the stack that is a part of activation record and the caller functions gets the value at that point from the stack and use it. In the above example, we have return value as a reference as *EType&*. Can a function return a reference of its local variable? When the function ends, the local variables are destroyed. So returning the reference of local variable is a programming mistake. Therefore, a function returns the reference of some member variable of the class. By not writing the & with the return type, we are actually returning the value of the variable. In this case, a copy of the returning variable is made and returned. The copy constructor is also used here to create the copy of the object. When we are returning by value, a copy is created to ascertain whether it is a local variable or member variable. To avoid this, we use return by reference. Now we want that the variable being returned, does not get changed by the calling function especially if it is the member variable.

These are the common usage of *const*. It is mostly used with the member function. It is just due to the fact that we avoid creating copy of the object and secondly we get our programming disciplined. When we send a reference to some function or get a reference from some function, in both cases while using the *const*, we guard our objects. Now these objects cannot be changed. If the user of these objects needs to change the object, he should use the *set* methods of the object.

AVL Tree

AVL tree has been named after two persons Adelson-Velskii and Landis. These two had devised a technique to make the tree balanced. According to them, an AVL tree is identical to a BST, barring the following possible differences:

- Height of the left and right subtrees may differ by at most 1.
- Height of an empty tree is defined to be (–1).

We can calculate the height of a subtree by counting its levels from the bottom. At some node, we calculate the height of its left subtree and right subtree and get the difference between them.

Lecture No. 20

AVL Tree: in the year 1962, two Russian scientists, Adelson-Velskii and Landis, proposed the criteria to save the binary search tree (BST) from its degenerate form. This was an
effort to propose the development of a balanced search tree by considering the height
as a standard. This tree is known as AVL tree. The name AVL is an acronym of the
names of these two scientists.
An AVL tree is identical to a BST, barring one difference i.e. the height of the left
and right sub-trees can differ by at most 1. Moreover, the height of an empty tree is
defined to be (–1).

Let’s consider a tree where the condition of an AVL tree is not being fulfilled. The
following figure shows such a tree in which the balance of a node (that is root node 6)
is greater than 1. In this case, we see that the left subtree of node 6 has height 3 as its
deepest nodes 3 and 5 are at level 3. Whereas the height of its right subtree is 1 as the
deepest node of right subtree is 8 i.e. level 1. Thus the difference of heights (i.e.
balance) is 2. But according to AVL definition, the balance should be1, 0 or –1.
From the above discussion, we encounter two terms i.e. height and balance which can
e defined as under.
Height
The height of a binary tree is the maximum level of its leaves. This is the same definition as of depth of a tree.
Balance
The balance of a node in a binary search tree is defined as the height of its left subtree minus height of its right subtree. In other words, at a particular node, the difference in heights of its left and right subtree gives the balance of the node.
Insertion of Node in an AVL Tree:
Now let’s see the process of insertion in an AVL tree. We have to take care that the tree should remain AVL tree after the insertion of new node(s) in it. We will now see ho an AVL tree is affected by the insertion of nodes.
We have discussed the process of inserting a new node in a binary search tree in previous lectures. To insert a node in a BST, we compare its data with the root node. If the new data item is less than the root node item in a particular order, this data item will hold its place in the left subtree of the root. Now we compare the new data item with the root of this left subtree and decide its place. Thus at last, the new data item becomes a leaf node at a proper place. After inserting the new data item, if we traverse the tree with the inorder traversal, then that data item will become at its appropriate position in the data items.

Lecture No. 21

While building an AVL tree, we rotate a node immediately after finding that that the node is going out of balance. This ensures that tree does not become shallow and remains within the defined limit for an AVL tree.

You are required to practice this inorder traversal. It is very important and the basic point of performing the rotation operation is to preserve the inorder traversal of the tree. There is another point to note here that in Binary Search

Tree (BST), the *root* node remains the same (the node that is inserted first). But in an AVL tree, the *root* node keeps on changing.

Cases of Rotation

The single rotation does not seem to restore the balance. We will re-visit the tree and rotations to identify the problem area. We will call the node that is to be rotated as a(node requires to be re-balanced). Since any node has at the most two children, and a height imbalance requires that a's two sub-trees differ by two (or –2), the violation will occur in four cases:

1. An insertion into left subtree of the left child of a.
2. An insertion into right subtree of the left child of a.
3. An insertion into left subtree of the right child of a.
4. An insertion into right subtree of the right child of a.

The insertion occurs on the *outside* (i.e., left-left or right-right) in *cases 1* and *4*. Single rotation can fix the balance in *cases 1* and*4*. Insertion occurs on the *inside* in *cases 2* and *3* which a single rotation cannot fix.

Lecture No. 22

C++ Code for avlInsert method: Now let's see the C++ code of avlinsert method. Now we have to include this balancing procedure in the insert method. We have already written this insert method which takes some value and adds a node in the tree. That procedure does not perform balancing. Now we will include this balancing feature in our insert method so that the newly created tree fulfills the AVL condition.

Here is the code of the function.

```
/* This is the function used to insert nodes satisfying the AVL condition.*/

TreeNode<int>*   avlInsert(TreeNode<int>* root, int info)
{
   if( info < root->getInfo() ){
       root->setLeft(avlInsert(root->getLeft(), info));
       int htdiff = height(root->getLeft()) – height(root->getRight());
       if( htdiff == 2 )
           if( info < root->getLeft()->getInfo() ) // outside insertion case
             root = singleRightRotation( root );
           else  // inside insertion case
             root = doubleLeftRightRotation( root );
  }
  else if(info > root->getInfo() ) {
       root->setRight(avlInsert(root->getRight(),info));
       int htdiff = height(root->getRight()) – height(root->getLeft());
       if( htdiff == 2 )
           if( info > root->getRight()->getInfo() )
               root = singleLeftRotation( root );
           else
               root = doubleRightLeftRotation( root );
   }
   // else a node with info is already in the tree. In
  // case, reset the height of this root node.
   int ht = Max(height(root->getLeft()), height(root->getRight()));
   root->setHeight( ht + 1 ); // new height for root.
   return root;
}
```

We have named the function as avlInsert. The input arguments are root node and the info is of the type int. In our example, we are having a tree of int data type. But of course, we can have any other data type. The return type of avlInsert method is TreeNode<int>*. This info will be inserted either in the right subtree or left subtree of root. The first if statement is making this decision by comparing the value of info and the value of root. If the new info value is less than the info value of root, the new node will be inserted in the left subtree of root. In the next statement, we have a recursive call as seen in the following statement.

avlInsert(root->getLeft(), info)

Here we are calling avlInsert method again. We are passing it the first node of the left subtree and the info value. We are trying to insert this info in the left subtree of root if it does not already exist in it. In the same statement, we are setting the left of root as the return of the avlInsert method. Why we are doing setLeft? As we know that this is an AVL tree and the new data is going to be inserted in the left subtree of root. We have to balance this tree after the insertion of the node. After insertion, left subtree may be rearranged to balance the tree and its root can be changed. You have seen the previous example in which node 1 was the root node in the start and in the end node 7 was its root node. During the process of creation of that tree, the root nodes have been changing. After the return from the recursive call in our method, the left node may be different than the one that was before insertion. The complete call is as:

root->setLeft(avlInsert(root->getLeft(), info));

Here we are inserting the new node and also getting the root of the left subtree after the insertion of the new node. If the root of the left subtree has been changed after the insertion, we will update our tree by assigning this new left-node to the left of the root.

Due to the insertion of the new node if we have rearranged the tree, then the balance factor of the root node can be changed. So after this, we check the balance factor of the node. We call a function height to get the height of left and right subtree of the root. The height function takes a node as a parameter and calculates the height from that node. It returns an integer. After getting the heights of left and right subtrees, we take its difference as:

int htdiff = height(root->getLeft()) – height(root->getRight());

Now if the difference of the heights of left and right subtrees is greater than 1, we have to rebalance the tree. We are trying to balance the tree during the insertion of new node in the tree. If this difference is 2, we will perform single rotation or double rotation to balance the tree. We have to deal with one of the four cases as discussed earlier. This rotation will be carried out with only one node where the balance factor is 2. As a result, the tree will be balanced. We do not need to rotate each node to balance the tree. This is characteristic of an AVL tree.

We have seen these rotations in the diagrams. Now we have to perform these ones in the code. We have four cases. The new node is inserted in the left or right subtree of the left child of a node or the new node is inserted in the left or right subtree of the right child of anode. How can we identify this in the code? We will identify this with the help of the info value of the new node.

We can check whether the difference (htdiff) is equal to 2 or not. We have if condition that checks that the info is less than the value of left node of the current node (a node). If this condition is true, it shows that it is the case of outside insertion. The node will be inserted as the left-most child. We have to perform single rotation. If the else part is executed, it means that this is the case of inside insertion. So we have to perform double rotation. We are passing a node to the rotation function. When a programmer performs these rotations, he gets another node that is assigned to root. This is due to the fact that the root may change during the rotations.

We encounter this case when the new node is inserted in the left subtree of the root. The new node can be inserted in the right subtree of the root. With respect to symmetry, that code is also similar. You can easily understand that. In the code, we have if-else condition that checks whether the info is greater than the value of root. Therefore, it will be inserted in the right subtree. Here again, we made a recursive call to avlInsert by passing it the first node of the right subtree of the root. When this recursive call is completed, we may have to perform rotations. We set the right node of the root as the node returned by the avlInsert. After this, we check the balance factor. We calculate the difference as:

int htdiff = height(root->getRight()) – height(root->getLeft());

In the previous case, we have subtracted the height of right tree from left tree. Here we have subtraction the height of left tree from the right tree. This is due to the fact that we want to avoid the -ve value. But it does not matter whether we get the -ve value. We can take its absolute value and test it. If the value of htdiff is 2 , we have to identify the case. We check that the new node is to be inserted in the right subtree of the right child of a node or in the left subtree of the right child of a node. If the info is greater than the value of the right child of root, it is the case 4. Here we will restore the balance of the tree by performing single rotation. If the else part is executed, we have to perform the double rotation. These rotation routines will return the root of the rearranged tree that will be set as a root.

In the end, we have the third case. This is the case when the info is equal to the root. It means that the data is already in the tree. Here we will readjust the height if it is needed. We will take the maximum height between the left and right subtree and set this height plus one as the height of the root. We have added one to the height after adding one node to the tree. In the end, we return the root. We have included the balancing functionality in our tree and it will remain balanced.

We have not seen the code of rotation routines so far. The code of these methods is also very simple. We have to move some of pointers. You can easily understand that code. We will discuss this in the next lecture. Please see the code of single and double rotation in the book and try to understand it.

FAQs

Question: What is an Array?

Answer: An array is a data structure that allows storage of a sequence of values. The values are stored in a contiguous block of memory. Arrays allow fast random access to particular elements. If the number of elements is indefinite or if insertions are required then we can't use an array. Example:

```
int idnumbers[100];
```

This declares an array of 100 integers named idnumbers.

Question: What is an Array Element?

Answer: A data value in an array.

Question: What is a Pointer?

Answer: In programming and information processing, a variable that contains the memory location (address) of some data rather than the data itself.

Question: What is Real Storage?

Answer: The amount of RAM memory in a system, as distinguished from virtual memory.Also called physical memory, physical storage.

Question: What is Priority Queue?

Answer: A priority queue is a specialized queue in which the items ares stored in order. A priority queue allows access to the smallest(or sometimes the largest)item.

Question: What is Stack?

Answer: A collection of items in which only the most recently added item may be removed. The latest added item is at the top. Basic operations are push and pop. Also known as "last-in, first-out" or LIFO.

Question: Difference between Stack and Queue?

Answer: Stack is "Last in first out" LIFO. Queue is "First in first out"FIFO

Question: What is Method?

Answer: A Method, or Member Function is a routine which is associated with a data structure to make a class. A Method can be invoked, and when it executes it has access to the data in the class, as well as data passed by way of arguments.

Question: What is an Algorithm?

Answer: An Algorithm is a generic term for any procedure. We usually mean a procedure that can be implemented as a routine, and which performs some well defined task. In general there are several possible Algorithms to perform the same task. NAO's Algorithm class is an organizational class that does note define any interface, but merely groups classes. Algorithms that perform the same task (or type of task) will share an interface defined in an abstract base class derived from the Algorithm class.

Question: What is an Interface?

Answer: The interface of a class is the set of routines which are provided to manipulate instances of the class.

Question: What is a Queue?

Answer: A multielement data structure from which (by strict definition) elements can be removed only in the same order in which they were inserted; that is, it follows a first-in-first-out (FIFO) constraint. The important queue operation are inserting an item at the rear of the queue and removing the item from the front of the queue.

Question: What is Abstract Data Type(ADT)?

Answer: In programming, a data set defined by the programmer in terms of the information it can contain and the operations that can be performed with it. An abstract data type is more generalized than a data type constrained by the properties of the objects it contains—for example, the data type "pet" is more generalized than the data types "pet dog," "pet bird," and "pet fish." The standard example used in illustrating an abstract data type is the stack, a small portion of memory used to store information, generally on a temporary basis. As an abstract data type, the stack is simply a structure onto which values can be pushed (added) and from which they can be popped (removed). The type of value, such as integer, is irrelevant to the definition. The way in which the program performs operations on abstract data types is encapsulated, or hidden, from the rest of the program. Encapsulation enables the programmer to change the definition of the data type or its operations without introducing errors to the existing code that uses the abstract data type. Abstract data types represent an intermediate step between traditional programming and object-oriented programming.

Question: What is difference between int* i and int *i?

Answer: There is no difference between int* i or int *i as far as the compiler is concerned. Both declare "i' to a pointer to an integer.The actual difference is how this declaration is perceived. The declaration syntax in general is "type variable", e.g., "int x", "double z", "char c" etc. If we follow this pattern, suppose we want the type of a variable "time" to be "pointer to int". To do so, we would write "int* time"; "int*" is the type and "time" is variable. However, the notation "*time" can be thought of as a variable name with the "*" included. [Variables names cannot being with a "*" normally]. To get to the actual integer stored in memory, the syntax is "*time", e.g., "*time = 10", "y = *time". In this approach, the type is "int" and the variable name is "*time" and the declaration thus "int *time".

Question: What is head?

Answer: Head: The first item of a list is called head.

Question: What is argc and argv?

Answer: C and C++ have a special argument list for main( ), which looks like this:

```
int main(int argc, char* argv[]) { // ...
```

The first argument is the number of elements in the array, which is the second argument. The second argument is always an array of char*, because the arguments are passed from the command line as character arrays (and remember, an array can be passed only as a pointer). Each whitespace-delimited cluster of characters on the command line is turned into a separate array argument. The following program prints out all its command-line arguments by stepping through the array:

Question: Difference between binary, unary and operand?

Answer:

Binary Operator: Any Mathematical operation which involves two operands for one operator e.g. A+B

Where A and B are operands and + is a binary operator.

Unary Operator: Characteristic of a mathematical operation with a single operand (object)

Operand: A quantity on which an operation is performed.

Question: Difference between .h file and .cpp file?

Answer: Header File: The subdirectory called INCULDE contains header files. These files (also called "include" files) are text files, like the ones you generate with a word processor or the Dev C++ Editor. Header files can be combined with your program before it is compiled, in the same way that a typist can insert a standard heading in a business letter. Each header file has .h file extension.

Header files serve several purposes. You can place statements in your program listing that are not program code but are instead messages to the compiler. These messages, called compiler directives, can tell the compiler such things as the definitions of words or phrases used in your program. Some useful compiler directives have been grouped together in header files, which can be included in the source code of your program before it goes to the compiler.

CPP File: Class methods are defined (implement) in the CPP files. Also main function is defined in the CPP file.

Question: What is difference between Pseudo code and Algorithm?

Answer: Pseudo code:Any informal, transparent notation in which a program or algorithm description is written. Many programmers write their programs first in a pseudocode that looks much like a mixture of English and their favorite programming language, such as C or Pascal, and then translate it line by line into the actual language being used.

Algorithm:A step-by-step problem-solving procedure, especially an established, recursive computational procedure for solving a problem in a finite number of steps.

Question: What is external node?

Answer: A terminal or "bottom" item of a tree,i.e.an item with no child known as external node.

Question: What is the formula for finding the minimum number of nodes in AVL tree i.e logon for BST?

Answer:

AVL Trees

similar to binary search trees

difference: for every node left and right subtrees can have height difference of at most 1

height of an AVL tree = at most 1.44 log n (approx.)

N(h): # of nodes in minimum size AVL tree of height h

$N(h) = N(h-1) + N(h-2) + 1$

$N(0) = 1, N(1) = 2$

all operations, except insertion, performed in O(log N) time.

Question: What is Data Structure?

Answer: A data structure is a way of grouping fundemental types (like integers, floating point numbers, and arrays) into a bundle that represents some identifiable thing. For example, a matrix may be thought of as the bundle of the number of rows and columns, and the array of values of the elements of the matrix. This information must be known in order to manipulate the matrix. C introduced the struct for declaring and manipulating data structures. C++ extended the struct to a class.

Question: Define Heap?

Answer: 1. A portion of memory reserved for a program to use for the temporary storage of data structures whose existence or size cannot be determined until the program is running. To build and use such elements, programming languages such as C and Pascal include functions and procedures for requesting free memory from the heap, accessing it, and freeing it when it is no longer needed. In contrast to stack memory, heap memory blocks are not freed in reverse of the order in which they were allocated, so free blocks may be interspersed with blocks that are in use. As the program continues running, the blocks may have to be moved around so that small free blocks can be merged together into larger ones to meet the program's needs.

2. A complete binary tree in which the value of any node is not exceeded by the value of either of its children.

Question: Name the Properties of a Binary Tree?

Answer: Since binary tree nodes can only have a maximum of two children, this fact introduces several properties that do not make sense on general trees. There are three important properties that a binary tree can have: fullness, balance, and leftness.

Question: Define Binary Search Trees (BST) with example?

Answer: The most popular variation of the Binary Tree is the Binary Search Tree (BST). BSTs are used to quickly and efficiently search for an item in a collection. Say, for example, that you had a linked list of 1000 items, and you wanted to find if a single item exists within the list. You would be required to linearly look at every node starting from the beginning until you

found it. If you're lucky, you might find it at the beginning of the list. However, it might also be at the end of the list, which means that you must search every item before it. This might not seem like such a big problem, especially nowadays with our super fast computers, but imagine if the list was much larger, and the search was repeated many times. Such searches frequently happen on web servers and huge databases, which makes the need for a much faster searching technique much more apparent. Binary Search Trees aim to Divide and Conquer the data, reducing the search time of the collection and making it several times faster than any linear sequential search.