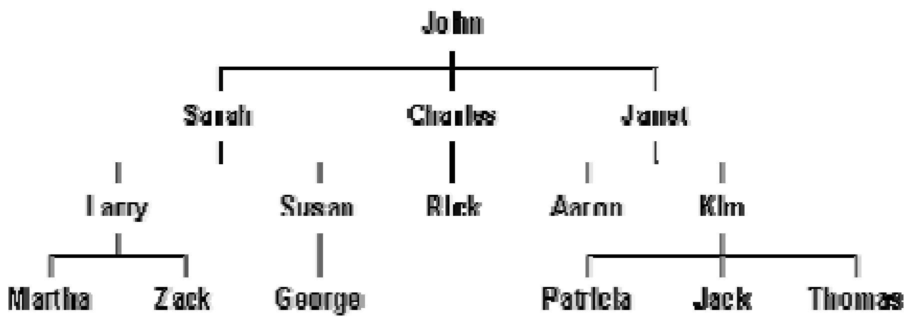
Introduction to Data Structures Using C

A data structure is an arrangement of data in a computer's memory or even disk storage. An example of several common data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables.

Algorithms, on the other hand, are used to manipulate the data contained in these data structures as in searching and sorting

Name	Position
Aaron	Manager
Charles	VP
George	Employee
Jack	Employee
Janet	VP
John	President
Kim	Manager
Larry	Manager
Martha	Employee
Patricia	Employee
Rick	Secretary
Sarah	VP
Susan	Manager
Thomas	Employee
Zack	Employee

But only one view of the company is shown by the above list. You also want the database to represent the relationships between employees and management at ABC. It does not tell you which managers are responsible for which workers and so on, although your list contains both name and position. After thinking about the problem for a while. You decide that the tree diagram is much better structure for showing the work relationships at the ABC company.



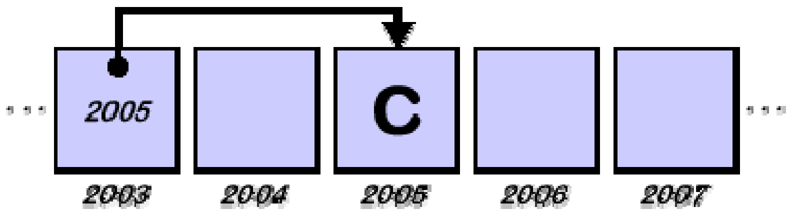
These two above diagrams are examples of the different data structures. Your data is organized into a list, in one of the data structures which is above. The employees name can be stored in alphabetical order so that we can locate the employee's record very quickly. For showing the relationships between employees these structure is not very useful. The tree structure is a much better suited for this purpose.

The data structures are an important way of organizing information in a computer. There are many different data structures that programmers use to organize data in computers, just like the above illustrated diagrams. Some of the data structures are similar to the tree diagram because they are good for representing the relationships between different data. Other structures are good for ordering the data in a particular way like the list of employees which is shown above. Each data structure has their own unique properties that make it well suited to give a certain view of the data.

Pointers and Indirection

The problem with representing Data Structures Using C that are not linear. We need some way to map these data structures to the computer's linear memory. One solution to this, is to use pointers.

Pointers are the memory locations that are stored in the memory cells. By using a pointer, by holding a memory address rather than the data one memory cell can point to another memory cell.



The memory cell at address 2003 contains a pointer the address of another cell, which can be seen in the above figure. Here the pointer is pointing to the memory cell 2005 which contains the letter C. This means that, we have two ways of accessing the letter C. We can refer to the memory cell which contains the value C directly or we can use the pointer to refer to it indirectly. The process of accessing the data through pointers is known as indirection. Using pointers, we can also create multiple levels of indirection.

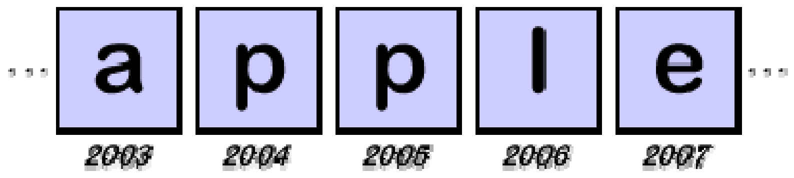
Linear Data Structures

Pointers can become very complex and difficult to use by having many levels of indirection. When used pointers incorrectly, it can make data structures very difficult to understand. The tradeoff between complexity and flexibility should be consider whenever you use pointers in constructing data structures.

The idea of pointers and indirection is not exclusive to the computer memory. Pointers appear in many different aspects of computer's usage. Hyperlinks in web pages is a good example pointers. This links are really the pointers to the other web page. Perhaps you have even experienced the double indirection when you went to visit the familiar web site and found the site had moved. You saw a notice that the web pages had been moved and a link to the new site, instead of the page you expected.

We saw that it is very simple to create data structures that are organized similar to the way the computer's memory is organized. For example, the list of employee's from the ABC company is a linear data structure.

Since the computer's memory is also linear, it is very easy to see how we can represent this list with the computer's memory. Any data structure which organizes the data elements one after the other is known as linear data structure. So far we have seen two examples of linear data structures: the string data structure and the ABC company list.



You may have noticed that these two examples of linear data structures resemble to each other. This is because they both are really different kinds of lists. In general, all linear data structures look like the list. However, this does not mean that all the linear data structures are exactly the same. Suppose I want to design a list to store the names of the ABC employees in the computer. One possible design is to organize the names similar to the example picture above. Another possible design is to use the pointers we learned about in the last lesson. While these two designs provide the same functionality the way they are implemented in the computer is much different. This means that there is an abstract view of a list which is distinct from any particular computer implementation

You may have also noticed that the picture of the ABC employees is not as exactly the same as the original list. When we make a list of names, we tend to organize this list into a column rather than a row. In this case, the conceptual or logical representation of a list is the column of names. However, the physical representation of the list in the computer's memory is the row of strings. For most data structures, the way that we think about them is far different from the way they are implemented in the computer. In other words, the physical representation is much different from that of the logical representation, especially in data structures that use pointers.

Ordered List: The Abstract View

The most common linear data structure used is the list. By now you are already pretty familiar with the idea of a list and at least one way of representing a list in the computer. Now we are going to look at a particular kind of list: an ordered list. Ordered lists are very similar to the alphabetical list of employee names for the ABC company. These lists keep items in a specific order such as alphabetical or numerical order. Whenever an item is added to the list, it is placed in the correct sorted position so that the entire list is always sorted.

Before we consider how to implement such a list, we need to consider the abstract view of an ordered list. Since the idea of an abstract view of a list may be a little confusing, let's think about a more familiar example. Consider the abstract view of a television. Regardless of who makes a television, we all expect certain basic things like the ability to change channels and adjust the volume. As long as these operations are available and the TV displays the shows we want to view, we really don't care about who made the TV or how they chose to construct it. The circuitry inside the TV set may be very different from one brand to the next, but the functionality remains the same. Similarly, when we consider the abstract view of an ordered list, we don't worry about the details of implementation. We are only concerned with what the list does, not how it does it.

Suppose we want a list that can hold the following group of sorted numbers: [2 4 6 7]. What are some things that we might want to do with our list? Well, since our list is in order, we will need some way of adding numbers to the list in the proper place, and we will need some way of deleting numbers we don't want from the list. To represent these operations, we will use the following notation:

```
AddListItem(List, Item)
RemoveListItem(List, Item)
```

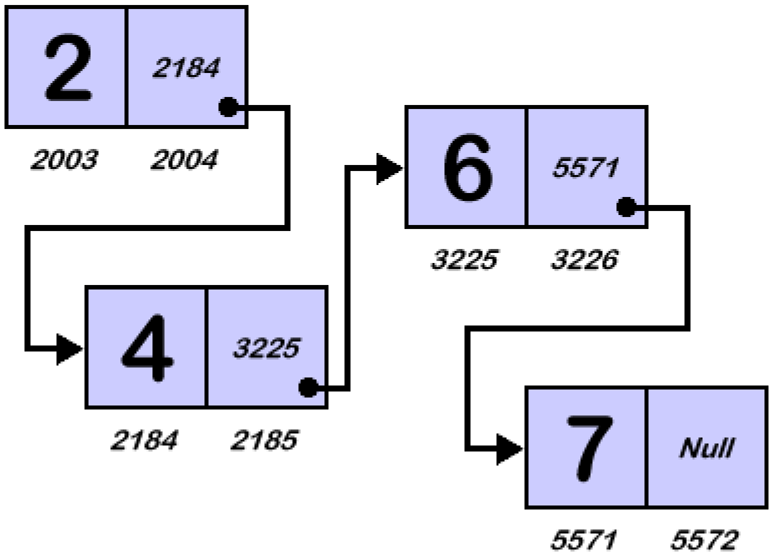
Each operation has the name and the list of parameters the operation needs. The parameter list for the AddListItem operation includes a list and an item. The RemoveListItem operation is very similar except this time we will specify the item we want to remove. These operations are part of the abstract view of an ordered list. They are what we expect from any ordered list regardless of how it is implemented in the computer's memory.

Array Implementation

One approach to creating a list is simply to reserve a block of adjacent memory cells to large enough to hold the entire list. Such a block of memory is called as array. Of course, since we want to add items to our list, we need to reserve more than just four memory cells. For now, we will make our array large enough to hold as much as six numbers.

Pointer Implementation

A second approach to creating a list is to link groups of memory cells together using the pointers. Each group of memory cells is called as a node. With this implementation every node contains the data item and the pointer to the next item in the list. You can picture this structure as a chain of nodes linked together by the pointers. As long as we know where the chain begins, we can follow the links to reach any item in the list. Often this structure is called as a linked list.

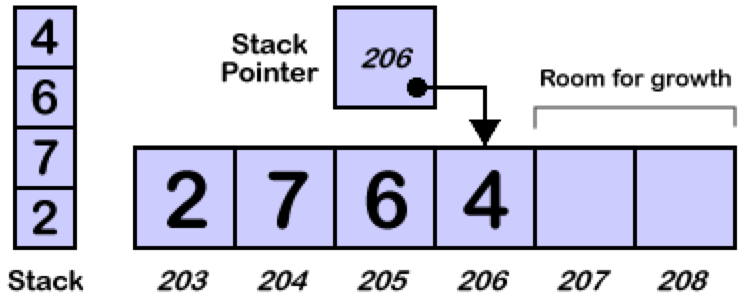


Notice that the last memory cell in our chain contains the symbol called "Null". This symbol is a special value that tells us that we have reached the end of our list. You can think that this symbol as a pointer that points to nothing. Since we are using the pointers to implement our list, the list operations AddListItem and the RemoveListItem will work differently than they did for sequential lists.

Linear Data Structures

Stack is the other common linear data structure which is been used now a days. Just like we did with the ordered list, we will examine the abstract view of the stack first and then look at the couple of ways a stack can be implemented.

Stack is very similar to a list except that a stack is more restricted. The figure below should give you an good idea of the abstract view of what stack is. Follow the directions to manipulate the simple stack and learn about the operations that the stack provides.



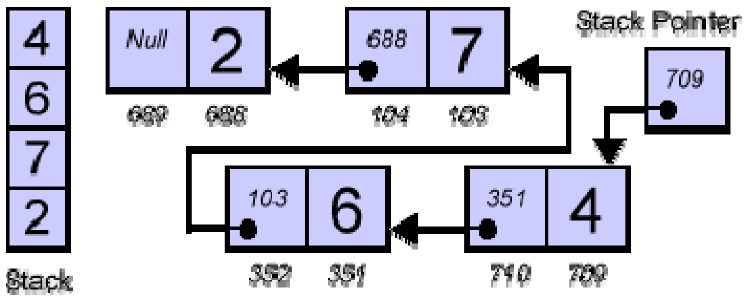
By seing the figure above you can see that this data structure is really a restricted list. You have restricted the access to one end of the list by using the pop and push operations. The result of this restriction is that items in the list will be stored one on top of the other. We must first remove all the items above it till you get to the bottom item. "Last-In, First-Out" or LIFO, which is used to describe the behavior, since the last item to enter the stack is the first item to leave the stack. The top item is the item always the last item to enter the stack and it is always the first item to leave the stack since no other items can be removed until the top item is removed.

```
PushStackItem(Stack, Item)
Item PopStackItem(Stack)
```

The PushStackItem operation has two parameters which are, the stack and an item. This operation adds the item to the top of the specified stack, the item and the stack which is specified in the above function. The PopStackItem operation only takes one parameter which is a stack(stack name). However, notice that this operation has the keyword Item listed to the left of the PopStackItem keyword. This keyword is used to represents the item that is removed from the top of the stack when the PopStackItem operation is done. These two operations(PushStackItem and PopStackItem) are part of the abstract view of the stack.

Pointer Implementation in the Stack

In order to implement a stack using pointers, we need to link nodes together just like we did for the pointer implementation of the list. Each node contains the stack item and the pointer to the next node. We also need a special pointer(stack pointer) to keep track of the top of our stack.



Stack Operations:

push(new-item:item-type)

Adds an item onto the stack.

top():item-type
Returns the last item pushed onto the stack.

pop()
Removes the most-recently-pushed item from the stack.

is-empty():Boolean
True iff no more items can be popped and there is no top item.

is-full():Boolean
True iff no more items can be pushed.

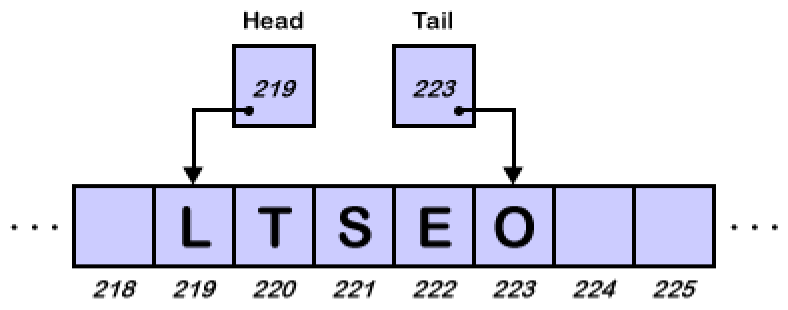
get-size():Integer
Returns the number of elements on the stack.

All operations except get-size() can be performed in *O*(1) time. get-size() runs in at worst *O*(*N*).

Queues

The Queue is the final linear data structure that we will examined. Like the stack, the queue is also a type of restricted list. Instead of restricting all the operations to only one end of the list as a stack does, the queue allows items to be added at the one end of the list and removed at the other end of the list. The figure below should give you a good idea of the abstract view of the queue.

This restrictions placed on a queue cause the structure to be a "First-In, First-Out" or FIFO structure. This idea is similar to customer lines at a in any bill payment store(eg. phone bill payment queue). When customer A is ready to check out, he or she enters the tail(end) of the waiting line. When the preceding customers have paid, then customer A pays and exits from the head of the line. The bill-payment line is really a queue that enforces a "first come, first serve" policy.



We will represent these two operations with the following notation:

```
EnqueueItem(Queue, Item)
Item DequeueItem(Queue)
```

These two operations are very similar to that of the operations we learned for the stack data structure. Although the names are different, the logic of using the parameters is the same. The EnqueueItem(enter the queue item) operation takes the Item parameter and adds it to the tail(end) of Queue. The DequeueItem(delete queue item) operation removes the head item of Queue and returns this as Item. Notice that we represent the returned item with a keyword located to the left of the operation name. These two operations are part of the abstract view of a queue. Regardless of how we choose to implement our queue on the computer, the queue must support these two operations.

The Implementation View

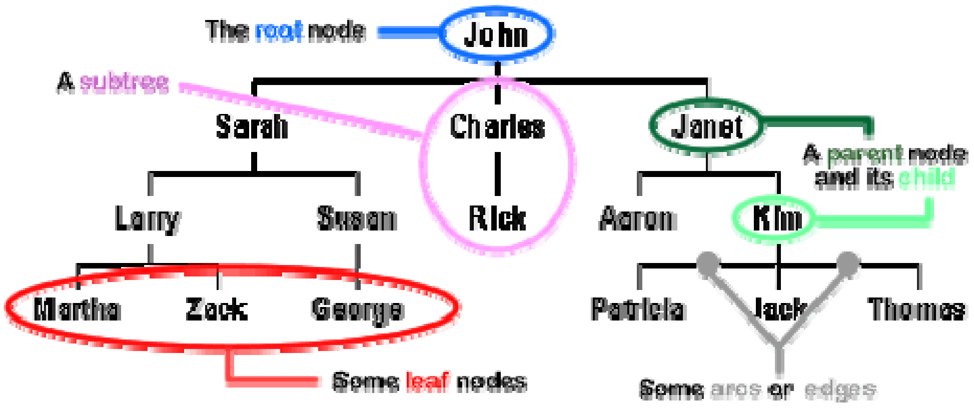
When we looked at the ordered list and stack data structures, we saw two different ways to implement each one of them. Although the implementations were different, the data structure was still the same from the abstract point of view for both stack and ordered list. We could still use the same operations on the data structures regardless of their implementations. With the queue, it is also possible to have various implementations that support the operations EnqueueItem and DequeueItem. The distinction between the logical representation of the queue and the physical representation of the queue. Remember that the logical representation is the way that we think of the way data being stored in the computer. The physical representation is the way the way data is actually organized in the memory cells(computer memory).

Now let's consider how the EnqueueItem and DequeueItem operations might be implemented in the queue. To store letters into the queue, we could advance the tail pointer by one location and add the new letter in the queue. To dequeue(delete) letters, we could remove out the head letter and increase the head pointer by one location. While this approach seems very straight forward, it has a serious problem. As items are added and removed, our queue will march straight through the entire memory of the computer. We have not limited the size of our queue to a fixed amount of size.

Perhaps we could limit the size of the queue by not allowing the tail pointer to advance beyond the certain memory location. This implementation would stop the queue from traversing the entire memory, but it would only allow us to fill the queue one time. Once the head and tail pointers reached the stop location, our queue would no longer work untill we delete some data from it.

Trees: The Abstract View

Another common nonlinear data structure is the tree.



In this diagram above, we can see that the starting point, or the root node, is circled in blue colour. A node is a simple structure that holds data and links to other nodes.

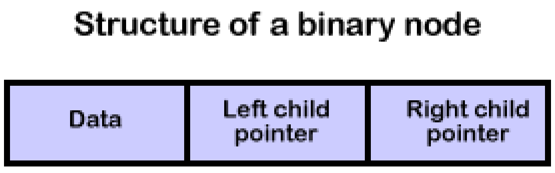
In this case, our root node contains the data string "John" and three links to the other nodes. Notice that the group of nodes circled in red do not have any links(childs). These nodes are at the end of the branches and they are appropriately called as leaves or leaf nodes. In our diagram, the nodes are interconnected with solid black lines called arcs or edges. These edges show the relationships between the nodes in the tree.

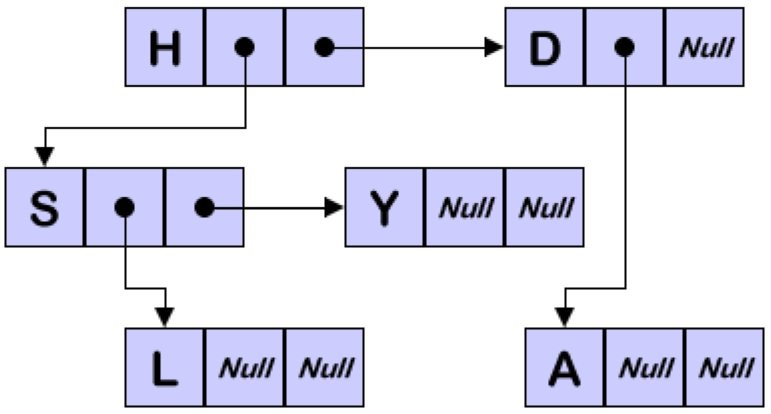
One important relationship in the binary tree is the parent-child relationship. Parent nodes have at least one edge to the node lower in the tree. This lower node is called the child node. Nodes can have more than one child, but the children can only have a single parent. Notice that the root node has no parent, and the leaf nodes has no children. Final feature to note in our diagram is the subtree. At each level of the tree, we can see that the tree structure is repeated. For example, the two nodes representing "Charles" and "Rick" compose a very simple tree with "Charles" as the root node and and "Rick" as a single leaf node.

Trees are implemented in the computer's memory. We will begin by introducing the simple tree structure called the binary tree. Binary trees have the restriction that nodes can't have more than two children. With this restriction, we can easily determine how to represent a single binary node in the memory. Our node will need to reserve memory for the data and two pointers(for pointing two childs of that node).

Structure of a binary node:

Using our binary nodes, we can construct a binary tree. In the data cell of each node, we will can store a letter. The physical representation of our tree might look something like the figure below:

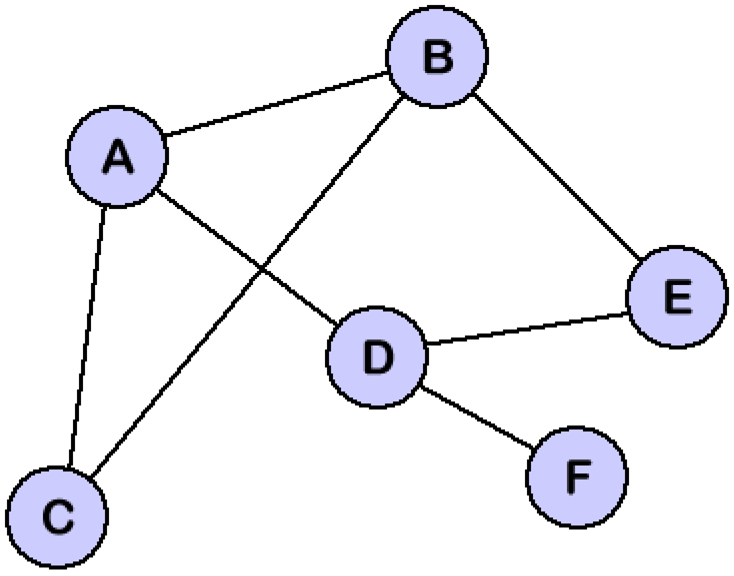




Introduction to Graphs

The last data structure that we will study in this tutorial is the graphs. Graphs are similar to trees except that, they do not have as many restrictions.

In the previous tutorial, we saw that every tree has a root node, and all the other nodes in the tree are children of a perticular node. We also saw that the nodes can have many children but only one parent. When we relax these restrictions, we get the graph data structure. The logical representation of a typical graph might look something like the figure shown below:



It is not hard to imagine how the graph data structure could be useful for representing the data. Perhaps each of the nodes above could represent a city and the edges connecting the nodes could represent the roads. Or we could use a graph to represent a computer network where the nodes are workstations and the edges are the network connections. Graphs have so many applications in the computer science and mathematics that several algorithms have been written to perform the standard graph operations such as searching the graph and finding the shortest path between nodes of a graph.

Notice that our graph does not have any root node like the tree data structure. Instead, any node can be connected with any other node in the graph. Nodes do not have any clear parent-child relationship like we saw in the tree. Instead nodes are called as neighbors if they are connected by an edge. For example, node A above has three neighbors: B, C, and D.

Now that you have a basic idea of the logical representation of the graphs, let's take a look at one way that graphs are commonly represented in computers. The representation is called an adjacency matrix, and it uses the two-dimensional array to store the information about the graph nodes. The adjacency matrix for our graph is given below.

	A	B	C	D	E	F
A	--	1	1	1	--	--
B	1	--	1	--	1	--
C	1	1	--	--	--	--
D	1	--	--	--	1	1
E	--	1	--	1	--	--
F	--	--	--	1	--	--

Notice that the matrix shown above has six rows and six columns labeled with the nodes from the graph. We mark a '1' in a cell if there exists an edge from two nodes that index that cell. For example, since we have a edge between A and B, we mark a '1' in the cells indexed by A and B. These cells are marked with a dark gray background in the adjacency matrix. With our adjacency matrix, we can represent every possible edge that our graph can have.

Quiz for Data Structures Using C

1. ______________ Memory is Volatile

✗ main

✗ Random Access

✓ Both 1 and 2

✗ Virtual

2. An __________ data type is a keyword of a programming language that specifies the amount of memory needed to store data and the kind of data that will be stored in that memory location

✓ abstract

✗ int

✗ vector

✗ None of these

3. Which of the following abstract data types are NOT used by Integer Abstract Data type group?

✗ Short

✗ Int

✓ float

✗ long

4. The hashString() member function is called by other member functions of the Hashtable class whenever a function needs to convert a ________________

✗ a hash number key to a key

✓ key to a hash number key

✗ a key to an Index

✗ None of these

5. An application iterates the hashtable by calling the ______ and ______ member functions

✗ hasNext() and hasDelete()

✓ hasNext() and getNextKey()

✗ Both 1 and 2

✗ None of these

6. The java.util package contains two classes that are designed to work with hashtables. They are _______ and _________..

✓ Hashtable , HashMap class

✗ Hashtable,List

✗ Vector,List

✗ Vector,Hashtable

7. Data members of the Hashtable class stored in the private access specifier

✓ private access specifier

✗ Public access specifier

✗ common access specifier

✗ None of these

8. _____ is the common programming technique used for hashing in all hashing functions

✗ Cloning

✓ Bit Shifting

✗ Hashmapping

✗ Listing

9. If the depth of a tree is 3 levels, then what is the Size of the Tree?

✗ 4

✗ 2

✓ 8

✗ 6

10. deleteNode() function requires the _____ of the data element of the node that is being removed

✗ reference

✓ value

✗ declaration

✗ variable

11. Value of the first linked list index is _______

✗ One

✓ Zero

✗ -1

✗ None of these

12. A linked list index is ____ that represents the position of a node in a linked list.

✓ An Integer

✗ a variable

✗ a character

✗ a boolean

13. Why is the constructor of the QueueLinkedList class empty?

✓ because initialization of data members of the LinkedList class is performed by the constructor of the LinkedList class.

✗ because initialization of data members of the LinkedList class is performed by the destructor of the LinkedList class.

✗ because initialization of data members of the QueueLinkedList class is performed by the constructor of the LinkedList class.

✗ because initialization of data members of the QueueLinkedList class is performed by the destructor of the LinkedList class

14. _______ form of access is used to add and remove nodes from a queue

✗ LIFO,Last In First Out

✓ FIFO , First In First Out

✗ Both 1 and 2

✗ None of these

15. ______ form of access is used to add and remove nodes from a stack

✓ LIFO

✗ FIFO

✗ Both 1 and 2

✗ None of these

16. New nodes are added to the _____ of the queue.

✗ front

✓ back

✗ middle

✗ Both 1 and 2

17. A _______ is a data structure that organizes data similar to a line in the supermarket, where the first one in line is the first one out.

✓ queue linked list

✗ stacks linked list

✗ both of them

✗ neither of them

18. In an array queue, data is stored in an _____ element.

✗ Node

✗ linked list

✓ array

✗ constructor

19. The pop() member function determines if the stack is empty by calling the _____ member function

✗ removeback()

✓ isEmpty()

✗ removedfront()

✗ hasNext()

20. What happens when you push a new node onto a stack?

✓ the new node is placed at the front of the linked list.

✗ the new node is placed at the back of the linked list.

✗ the new node is placed at the middle of the linked list.

✗ No Changes happens

```
// Implementation Of BINARY TREE OPERATION (insertion & Deletion)

#include<conio.h>
#include<stdio.h>
#include<malloc.h>
#include<process.h>

struct node
{
 struct node *llink;
 struct node *rlink;
 int data;
};
```

```
void main()
{
 struct node *head,*t;
 int s,d;
 struct node* finsert();
 struct node* delenode(struct node*,int);
 void insert(struct node *);
 void inorder(struct node *);
 clrscr();
 head=NULL;
 do
 {
  printf("
1-Insertion");
  printf("
2-Deletion");
  printf("
3-Inorder");
  printf("
4-Exit");
  printf("
Enter Choice:");
  scanf("%d",&s);
  switch(s)
  {
   case 1://insertion
     if(head==NULL)
      {
          head=finsert();
      }
     else
          insert(head);
     break;

   case 2://Deletion
     if(head==NULL)
       printf("
Binary Tree Empty.......");
     else
     {
       printf("
Enter Data to delete:");
       scanf("%d",&d);
       if(head->llink==NULL && head->rlink==NULL && head->data==d)
            {
             t=head;
```

```
              head=NULL;
              free(t);
            }
       else
              head = delenode(head,d);
     }
     break;

   case 3://to display
     printf("
IN-ORDER:");
     if(head==NULL)
      printf("
Binary Tree Empty....");
     else
      inorder(head);
     break;

   case 4://exit
     exit(0);
  }
 }while(s<5 ||s>0);
 getch();
}

struct node* finsert()
{
 struct node * head;
 head=(struct node*)malloc(sizeof(struct node));
 printf("
Enter Data:");
 scanf("%d",&head->data);
 head->llink=NULL;
 head->rlink=NULL;
 return head;
}

void insert(struct node *head)
{
 struct node *t,*n;
 t=head;
 n=(struct node *)malloc(sizeof(struct node));
 printf("
Enter Data:");
 scanf("%d",&n->data);
 n->llink=NULL;
 n->rlink=NULL;
 while(t->llink!=NULL || t->rlink!=NULL)
 {
  if(t->llink!=NULL)
    if(n->data < t->data)
      t=t->llink;
  if(t->rlink!=NULL)
    if(n->data>=t->data)
      t=t->rlink;
  if((t->llink==NULL) && (n->data < t->data) && (n->data <
t->rlink->data))
    break;
  if((t->rlink==NULL) && (n->data >= t->data) && (n->data >
t->llink->data))
    break;
 }
 if((n->data < t->data) && (t->llink==NULL))
  t->llink=n;
 if((n->data > t->data) && (t->rlink==NULL))
  t->rlink=n;
}

void inorder(struct node * head)
{
 if(head!=NULL)
 {
  inorder(head->llink);
  printf("%d     ",head->data);
  inorder(head->rlink);
 }
}

struct node * delenode(struct node *head,int d)
{
 int f=0,f1=0;
 struct node *p,*t,*t1,*x;
 t=head;

 //to search found or not
 while(t!=NULL)
 {
  if(t->data==d)
  {
   f=1;
   x=t;
   break;
  }
  if(t->data > d)
  {
   p=t;
   t=t->llink;
  }
  else if(t->data <= d)
  {
   p=t;
   t=t->rlink;
  }
 }
 if(f==0)
  {
    printf("
Given element not found.......");
    return head;
  }

 //Deleted node has no child
 if(x->llink==NULL && x->rlink==NULL)
 {
  if(p->rlink==x)
    p->rlink=NULL;
  else
    p->llink=NULL;
  free(x);
  return head;
 }

 //deleted node has 2 children
 if(x->llink!=NULL && x->rlink!=NULL)
 {
  p=x;
  t1=x->rlink;
  while(t1->llink!=NULL)
  {
    p=t1; f1=1;
    t1=t1->llink;
  }
  if(t1->llink==NULL && t1->rlink==NULL)
  {
   x->data=t1->data;
   if(f1==1)
     p->llink=t1->llink;
   if(f1==0)
```

```
    x->rlink=t1->rlink;
  free(t1);
  return head;
 }
 if(t1->rlink!=NULL)
 {
  x->data=t1->data;
  if(f1==1)
    p->llink=t1->rlink;
  if(f1==0)
    p->rlink=t1->rlink;
  free(t1);
  return head;
 }
}

//Deleted node has oniy right child
if(x->llink==NULL && x->rlink!=NULL && x->data!=head->data)
{
 if(p->llink==x)
  p->llink=x->rlink;
 else
  p->rlink=x->rlink;
 free(x);
 return head;
}

//Deleted node has oniy left child
if(x->llink!=NULL && x->rlink==NULL && x->data!=head->data)
{
 if(p->llink==x)
  p->llink=x->llink;
 else
  p->rlink=x->llink;
 free(x);
 return head;
}
if(x->llink!=NULL && x->rlink==NULL && x->data==head->data)
{
 head=x->llink;
 free(p);
 return head;
}
if(x->llink==NULL && x->rlink!=NULL && x->data==head->data)
{
 head=x->rlink;
```

```
  free(p);
  return head;
 }
}
```