Course Booklet for Data Structure Module

$$\begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix} + \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ \vdots \\ a_n + b_n \end{bmatrix}$$

Is it C or Logic?

Know how of Data Structures

By Emertxe

Version 3.0 (December 18, 2014)

(http://www.emertxe.com)
Course Email: embedded.courses@emertxe.com

Contents

0.1 Before we begin

0.1.1 Course: Education goals and objectives

This course is intended to make you able in critical thinking, problem solving and information literacy. You all have to identify a problem and analyze it in terms of its significant parts and the information needed to solve it as part and curriculam of this course.
Simply objectives are as below:

- Familiarize the student with the issues of Time complexity and examine various algorithms from this perspective.
- Familiarize the student with good programming design methods, particularly Top Down design.
- Develop algorithms for manipulating stacks, queues, linked lists, trees.
- Develop the data structures for implementing the above algorithms.
- Develop recursive algorithms as they apply to trees.

0.1.2 Student Learning Outcomes

- *Critical Thinking and Problem Solving:* Use skills for analysis of programming problems and selection of algorithms.
- *Computation:* Use mathematical skills to develop algorithms and verify program outputs.
- *Technology:* Select and use appropriate programming constructs to solve problems.
- *Information Literacy:* Use textbook, programming references and online help to access necessary information.

0.1.3 At the end of the course the student will:

- Learn, and become comfortable with, advanced C techniques
- Learn advanced data abstraction features of C, generic pointers, function pointers etc.
- Master advanced problem solving techniques such as recursion and lateral thinking.
- Study and make use of data structures such as linked lists, stacks, queues, trees.
- Study and compare algorithms such as sorts.

0.1.4 Collaboration Policy

Collaboration on assignments is acceptable, although you must write the code for your programs entirely by yourself. You must also acknowledge the people you worked with on an assignment.
If your program includes code that you obtained from another source, please acknowledge it. Specifically:

- You must compose your own solution to each assignment. You may discuss strategies for approaching the programming assignments with your classmates and you may receive general debugging advice from them, but you must write all your own code.
- You may not write a program together and turn in two copies of the same code.
- You may not copy another student's code.
- If you work with another student, you must acknowledge that student on your assignment. This acknowledgement includes date, time, and the nature of your discussion. You must be specific.
- You may borrow code from textbooks or from lecture material, as long as you cite your sources.

0.1.5 Late Assignment Policy

All assignments are due in class on the due date. Late assignments will be accepted, with a penalty of 10% off the grade for each day after the due date. Some assignments will have stricter late penalties if they are due close to an examination date, so that solutions may be posted before the exam. No assignments will be accepted once the assignment has been returned to the class.

0.1.6 Course flow

This course is divided into parts as:

- Part 0:

  At the conclusion of part 0 the student should be able to:

  - *Describe Time-complexity issues* - definitions of Big-OH, Running-time.
  - Analyze several previously defined algorithms to determine their running time and the order of their running time.

  Lab: Assignment and allotment of projects.

- Part 1:

  At the conclusion of part 1 the student should be able to describe the following in detail:

  - The algorithms for manipulating singly, doubly, and circular Linked Lists.
  - The Implementation of Linked Lists using an array and pointer variables.

  Lab: Students will begin coding first project.

- Part 2:

  At the conclusion of part 2 the student should be able to describe the following in detail:

  - The algorithms for manipulating stacks and queues.
  - The Implementation of the above using an array and Linked Lists.
  - Apply stacks to parsing and recursion problems.
  - Unfold the recursive program by coding it non recursively.

  Lab: Students will keep coding first project and all the assignment of previous classes should be done and submitted.

- Part 3:

  At the conclusion of part 3 the student should be able to :

  - Understand Algorithms for simple sorts and for best sorts.
  - Discuss algorithms for searching-hashing algorithm, binary and linear search.

  Lab:: Submit first project and begin to design and code second project.

- Part 4:

  At the conclusion of part 4 the student should be able to describe the following in detail:

  - Tree definitions.
  - Algorithms for tree traversals, insertions, deletions.
  - The Implementation of trees using pointer variables and arrays.

  Lab: Continue coding second project.

- Part 5:

  At the conclusion of part 5 the student should be able to describe the following in detail:

  - Algorithms for creating complete Binary trees and almost complete Binary trees.
  - Algorithms for Binary Search trees.
  - The Implementation of the above.

  Lab:: Submit first project and begin to design and code second project.

- Part 6:

  - Hand in Final Project

  NOTE:
  Each project should consist of the following:

  - *Program listing* - Including liberal use of comments and contiguously, a run of the project.
  - *Project design* - The top-down structure of the project with brief pseudo code describing the logic used in the program.
  - All the above submitted in a folder in a format specified by mentor.

Best of Luck!

Chapter 1

Introduction

1.1 Data structures? Does it have importance?

NOTES:

1.1.1 Abstract Data Types - ADT

A set of data values and associated operations that are precisely specified independent of any particular implementation. i.e. stack, queue, priority queue.

NOTES:

1.1.2 Data Structures

The term data structure refers to a scheme for organizing related pieces of information. The basic types of data structures include: files, lists, arrays, records, trees, tables.

Each of these basic structures has many variations and allows different operations to be performed on the data. A data structure is the concrete implementation of that type, specifying how much memory is required and, crucially, how fast the execution of each operation will be. However for most purposes the terms ADT and data structure are interchangeable, so don't worry too much about understanding the differences between them.

NOTES:

1.2 Timing

Every time we run the program we need to estimate how long a program will run since we are going to have different input values so the running time will vary. Since the running time will vary, we need to calculate the worst case running time. The worst case running time represents the maximum running time possible for all input values. We call the worst case timing **big Oh** written **O(n)**. The *n* represents the worst case execution time units.

How many time units each kind of programming statement will take:

- Simple programming statement:

  Example:

  ```
  k++;
  ```

  Complexity: *O(1)*

  Simple programming statements are considered 1 time unit.

  NOTES:

- Linear *for* loops:

  Example:

  ```
  k=0;
  for(i=0; i<n; i++)
   k++;
  ```

  Complexity: *O(n)*

  for loops are considered *n* time units because they will repeat a programming statement *n* times. The term linear means the *for* loop increments or decrements by 1

  NOTES:

- Non linear loops:

  Example:

```
k=0;                          |      k=0;
for(i=n; i>0; i=i/2)          |      for(i=0; i<n; i=i*2)
  k++;                        |        k++;
```

  Complexity: *O(log n)*

  For every iteration of the loop counter i will divide by 2. If i starts is at 16 then then successive *i's* would be 16, 8, 4, 2, 1. The final value of k would be 4. Non linear loops are logarithmic. The timing here is definitely $log_2 n$ because $2^4 = 16$. Can also works for multiplication.

  NOTES:

- Nested *for* loops:

  Example:

```
k=0;
for(i=0; i<n; i++)
        for(j=0; j<n; j++)
                k++;
```

  Complexity: $O(n^2)$: $O(n) * O(n) = O(n^2)$

  Nested *for* loops are considered n^2 time units because they represent a loop executing inside another loop. The outer loop will execute n times. The inner loop will execute n times for each iteration of the outer loop. The number of programming statements executed will be $n * n$.

  NOTES:

- Sequential *for* loops:

  Example:

```
k=0;
for(i=0; i<n; i++)
k++;
k=0;
for(j=0; j<n; j++)
    k++;
```

  Complexity: *O(n)*

  Sequential *for* loops are not related and loop independently of each other. The first loop will execute n times. The second loop will execute n times after the first loop finished executing. The worst case timing will be: *O(n) + O(n) = 2 * O(n) = O(n)* We drop the constant because constants represent 1 time unit. The worst case timing is *O(n)*.

  NOTES:

- Loops with non-linear inner loop:

  Example:

```
k=0;
for(i=0;i<n;i++)
    for(j=i; j>0; j=j/2)
            k++;
```

  Complexity: *O(n log n)*

  The outer loop is *O(n)* since it increments linear. The inner loop is *O(n log n)* and is non-linear because decrements by dividing by 2. The final worst case timing is: *O(n) * O(log n) = O(n log n)*

  NOTES:

- Inner loop incrementer initialized to outer loop incrementer:

  Example:

```
k=0;
for(i=0;i<n;i++)
    for(j=i;j<n;j++)
            k++;
```

  Complexity: $O(n^2)$

  In this situation we calculate the worst case timing using both loops. For every i loop and for start of the inner loop j will be *n-1 , n-2, n-3.*

  NOTES:

- Power loops:

  Example:

```
k=0;
k = 0;
for(i=1; i<=n; i=i*2)
     for(j=1; j<=i; j++)
        k++;
```

  Complexity: $O(2^n)$

  To calculate worst case timing we need to combine the results of both loops. For every iteration of the loop counter i will multiply by 2. The values for j will be 1, 2, 4, 8, 16 and k will be the sum of these numbers 31 which is 2^n *- 1.*

  NOTES:

- *if-else* statements:

  With an if else statement the worst case running time is determined by the branch with the largest running time.

  Example:

```
/* O(n) */
 if (x == 5)
      {
      k=0;
      for(i=0; i<n; i++)
        k++;
      }
 /* O(n2) */
 else
     {
     k=0;
     for(i=0;i<n;i++)
         for(j=i; j>0; j=j/2)
             k++;
     }
```

  Complexity: The largest branch has worst case timing of $O(n^2)$

  NOTES:

- Recursive:

  From our recursive function let *T(n)* be the running time. Recursion behaves like a loop.The base case is the termination for recursion.

  Example:

```
int f(int n)
        {
        if(n == 0)
               return 0;
        else
               return f(n-1) + n
        }
```

  Complexity:

  For the line: *if(n == 0)* return 0; this is definitely: *T(1)*
  For the line: else return *f(n-1) + n* the time would be :*T(n-1) + T(1)*
  The total time will be: *T(1) + T(n-1) + T(1) = T(n-1) + 2* which is *O(n)*.

  The lower the time complexity of an algorithm, the faster the algorithm will carry out the work in practice. apart from time complexity, space complexity is also important. This is essentially the number of memory cells which an algorithm needs. A good algorithm keeps this number as small as possible.

  There is often a time-space trade off in a problem, ie, it cannot be solved with low computing time AND low memory consumption. One then has to make a compromise and exchange computing time for memory cells which an algorithm needs or vice versa. Depending on which algorithm one chooses and how one parameterizes it.Hash tables have a very good time complexity at the expense of using more memory than is needed by other algorithms.

  NOTES:

1.3 Complexity Examples:

What is "big Oh" ? for:

- *1:*

```
for(i=0;i<n*n; i++)
      {
       for(j=i; j<n; j++)
           k++;
      }
```

NOTES:

- *2:*

```
for(i=0; i<n; i++)
       {
        for(j=i; j>0; j=j/2)
            k++;
       }
```

NOTES:

- *3:*

```
for(i=0; i<n; i=i*2)
       {
        for(j=i; j<n; j*j)
            k++;
       }
```

NOTES:

1.4 Difference between concepts and implementation

1.5 Stages in program design

- Identify the data structures

- Operations - Algorithms

- Efficiency (Complexity)

- Implementation
  - What goes into header files?

  - What goes into C program?

- What are libraries? Why do we need them?

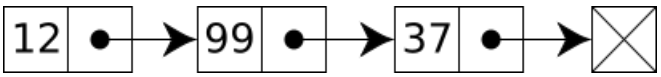
- How to create libraries?

Chapter 2

Linked List

2.1 Abstract

A collection of items accessible one after another beginning at the head and ending at the tail is called a list. A list implemented by each item having a link to the next item is a typical link list. A linked list arranges the data by logic rather than by physical address (as in arrays). The first item, or head, is accessed from a fixed location, called a "head pointer." An ordinary linked list must be searched with a linear search. A linked list can be used to implement other data structures, such as a queue or a stack. Linked lists are dynamic data structure, size is not fixed at compile time.



2.2 Linked list??? why...???

2.2.1 Linked lists vs. arrays

- Elements can be inserted into linked lists indefinitely, while an array will eventually either fill up or need to be resized.
- Further memory savings can be achieved.
- simple example of a persistent data structure.
- On the other hand, arrays allow random access, while linked lists allow only sequential access to elements.
- Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or boolean values.

2.3 Types of Link List

2.3.1 Linearly-linked list

- Singly-linked list

  The simplest kind of linked list is a singly-linked list (or slist for short), which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.

- Doubly-linked list

  A variant of a linked list in which each item has a link to the previous item as well as the next. This allows easily accessing list items backward as well as forward and deleting any item in constant time.also known as two-way linked list, symmetrically linked list.

2.3.2 Circularly-linked list

A variant of a linked list in which the nominal tail is linked to the head. The entire list may be accessed starting at any item and following links until one comes to the starting item again.

- Singly-circularly-linked list

  Similar to an ordinary singly-linked list, except that the next link of the last node points back to the first node.

- Doubly-circularly-linked list

  Similar to a doubly-linked list, except that the previous link of the first node points to the last node and the next link of the last node points to the first node.

2.4 Tradeoffs

2.4.1 Doubly-linked vs. singly-linked

Double-linked lists require more space per node, and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions.

2.4.2 Circularly-linked vs. linearly-linked

Allows quick access to the first and last records through a single pointer (the address of the last element). Their main disadvantage is the complexity of iteration, which has subtle special cases.

2.5 Refresh

2.5.1 Pointers:

- Pointer / Pointee:
- Dereference
- Bad Pointer
- Pointer Arithmatics
- Dynamic memory allocation / deallocation
- NULL

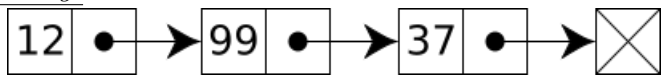
2.5.2 Structures:

- Templates
- *sizeof()* structure
- self referential structure

2.6 Drawing: Best way to design

2.7 Singly Linked List

Drawing:



Algo:

Our node data structure will have two fields. We also keep a variable HeadNode which always points to the first node in the list, or is null for an empty list.

```
struct Node {
        datatype data;  // The data being stored in the node
        Node next // A reference to the next node, null for last node
}
Node HeadNode  // points to first node of list;null for empty list
```

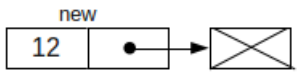
Traversal of a singly linked list is simple, beginning at the first node and following each next link until we come to the end:

```
temp = HeadNode
while temp not null
 (do something with temp->data)
  temp := temp->next
```

2.7.1 Operations on Linked Lists

- Create a new node

  Drawing:



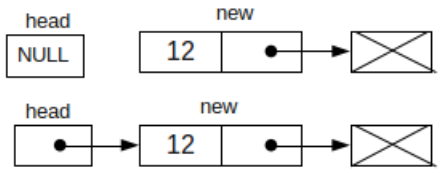
  Code:

```
struct linkedlist
{
        int data;
        struct linkedlist *next;
};
typedef struct linkedlist SList;

SList *head = NULL;  //points to first node, now stores NULL

SList* createnode (int element)
{
        SList *new = NULL;

        new = (SList*) malloc (sizeof (SList)); //allocates node
        if (new == NULL)
        {
                //error , memory not allocated
                return NULL;
        }
        new -> data = element;
        new -> next = NULL;

        return new;
}
```

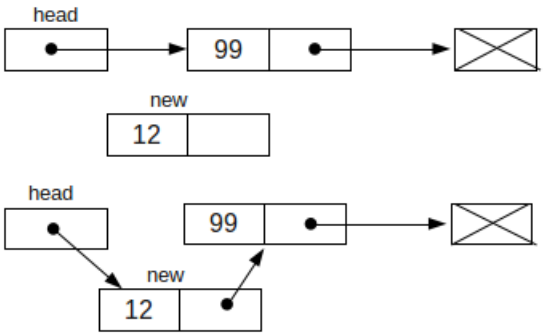
- Insert an element
  - Insert first

    Drawing:

    Insert first if list is empty (head is NULL)



    Insert first if list is not empty (head contains address of first node)

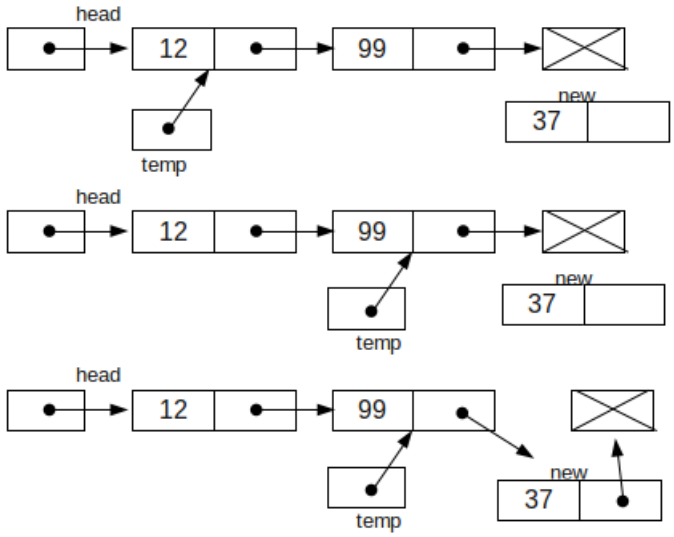


    Algo:

```
newnode->next = head
head = new
```

– Insert last

Drawing:



temp is traversed till it reaches the last node.

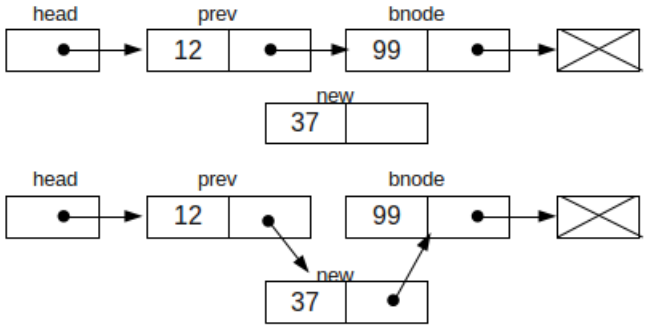
Algo:

```
temp -> next = new
new -> next = NULL
```

– Insert before a given element

Drawing:

Insert a node with value 37 before the node with value 99



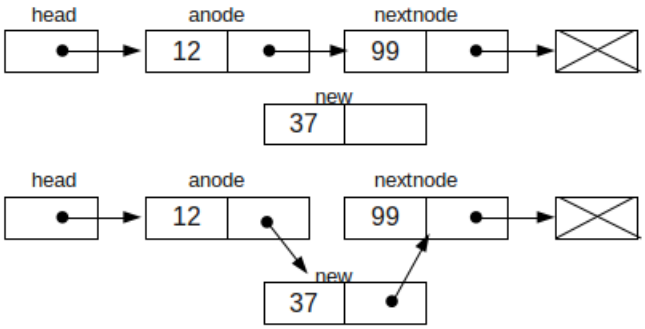
Algo:

```
prevnode->next = newnode
newnode->next = bnode
```

- Insert after a given element

  Drawing:

  Insert a new node with data 37 after the node with data 12



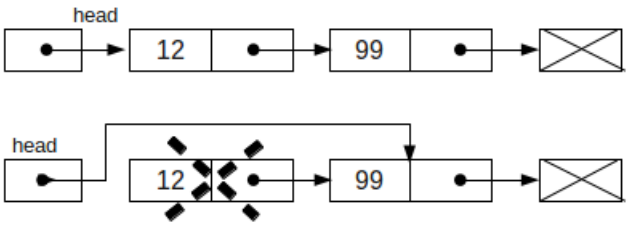
  Algo:

```
anode->next = newnode
newnode->next = nextnode
```

- Delete an Element

  - Delete first

    Drawing:



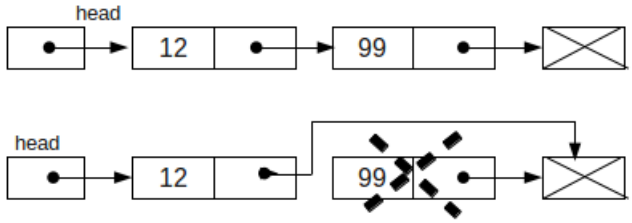
    Algo:

```
deletenode = head
```

```
head = head -> next
free (deletenode)
```

– Delete last

Drawing:

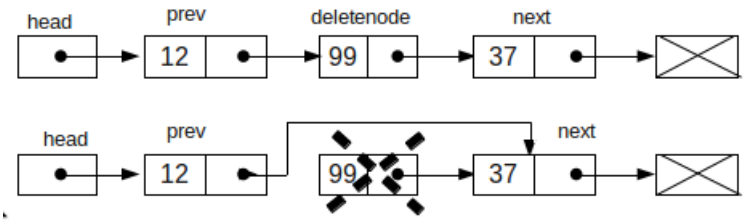


Algo:

```
secondlastnode->next = NULL
free (lastnode)
```

– Delete element

Drawing:

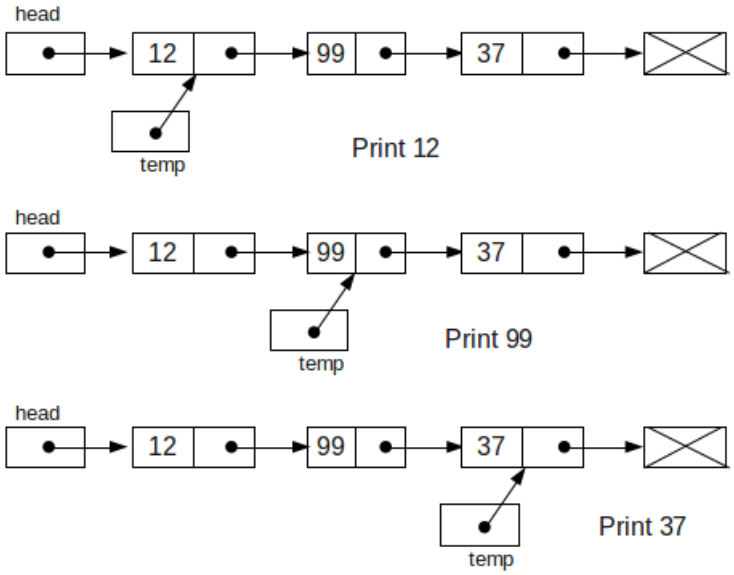


Algo:

```
prevnode->next = nextnode
free (deletenode)
```

- Print the list

  Drawing:



  Algo:

```
tempnode = head
while (tempnode not null)
{
        print temp->data
        temp = temp->next
}
```

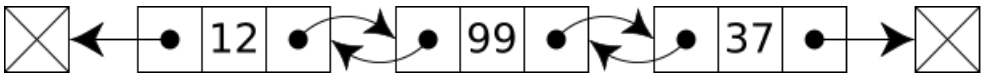
- Destroy the list

  Drawing:

  Algo:

  Code:

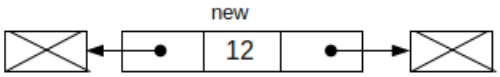
2.8 Doubly Linked List




```
typedef struct double_linkedlist {
        datatype data;
        struct double_linkedlist *prev, *next;
}DList;

DList *head = NULL;  //points to first node, now stores NULL
```

- Create a new node

  Drawing:



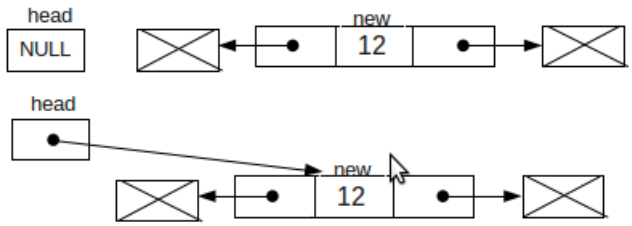
  Code:

```
DList* createnode (int element)
{
        DList *new = NULL;

        new = (DList*) malloc (sizeof (DList)); //allocates node
        if (new == NULL)
        {
        //error , memory not allocated
        return NULL;
        }
        new -> data = element;
        new -> next = NULL;
        new -> prev = NULL;

        return new;
}
```

- Insert first

  If list is empty (head is NULL)

  Drawing:

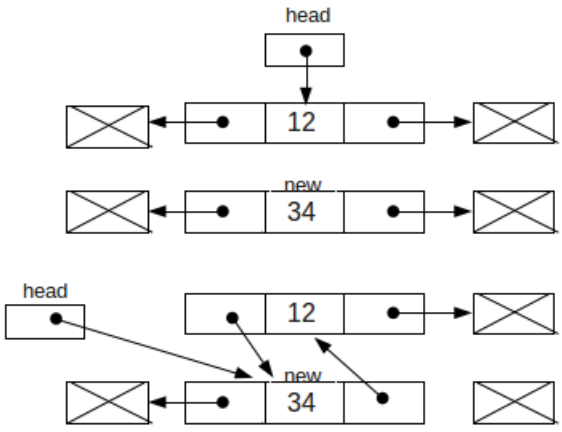


  Algo:

```
newnode->prev = head
newnode->next = head
head = newnode
```

  If list is not empty (head is not NULL)

  Drawing:

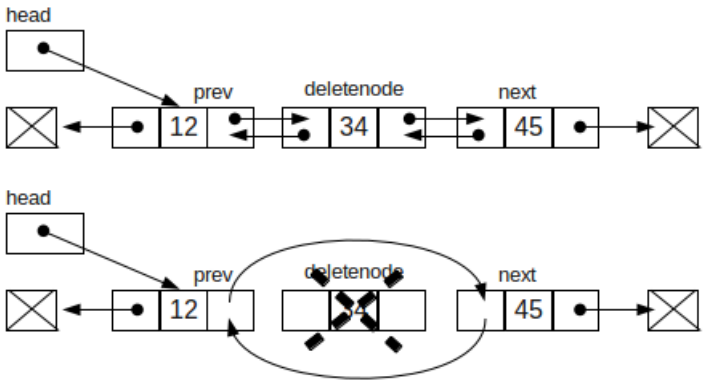


  Algo:

```
head->prev = newnode
newnode->next = head
head = newnode
```

- Delete element

  Drawing:

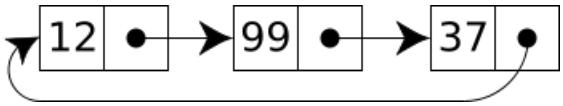


  Algo:

```
prevnode->next = nextnode
nextnode->prev = prevnode
free (deletenode)
```

2.8.1 Applications

2.9 Circular Linked List




```
lastnode -> next = headnode
```

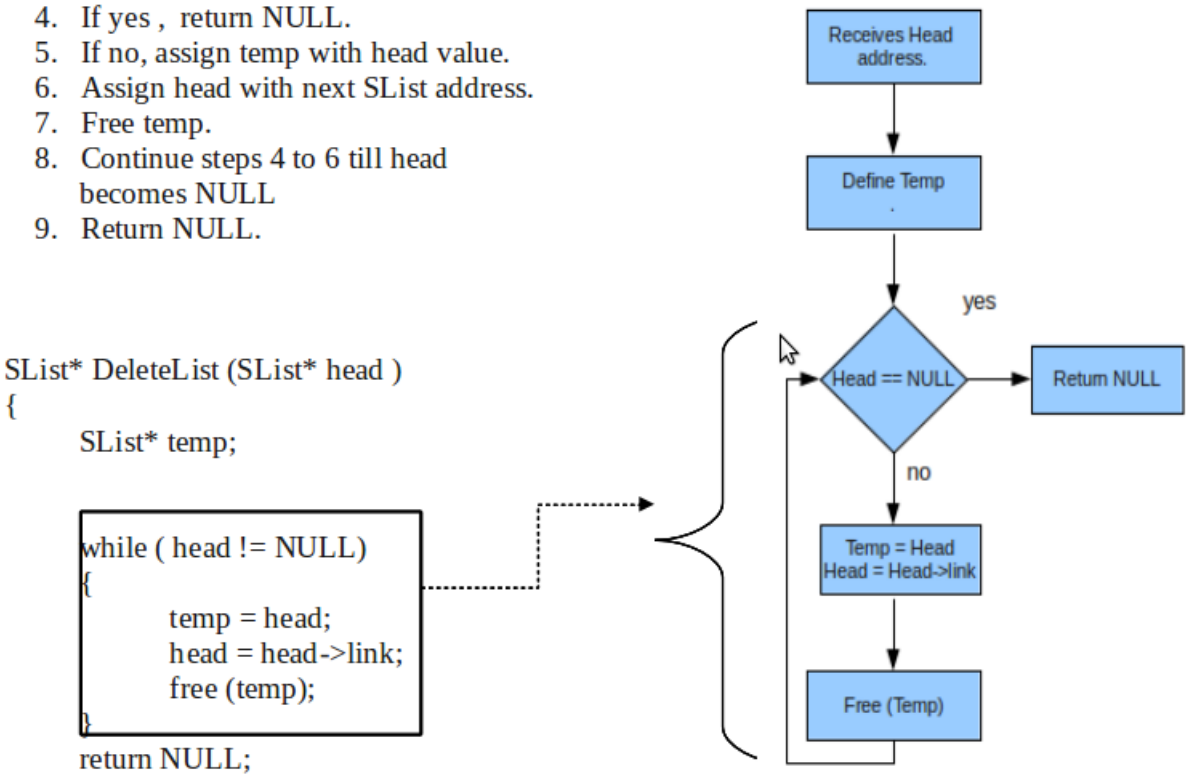
2.10 Lab Work

2.10.1 Practice

Write a function DeleteList that takes a list, deallocates all of its memory and sets its head pointer to NULL (the empty list).

DeleteList Function

1. Function passes the head pointer
2. Define a temp variable.
3. Check If head is NULL.
4. If yes , return NULL.
5. If no, assign temp with head value.
6. Assign head with next SList address.
7. Free temp.
8. Continue steps 4 to 6 till head becomes NULL
9. Return NULL.




```
SList* DeleteList (SList* head )
{
        SList* temp;

        while ( head != NULL)
        {
                temp = head;
                head = head->link;
                free (temp);
        }
        return NULL;
}
```

2.10.2 List of Assignments

(Id) / Date	Assignment Topic

( )	Create a library file named slist.c and include all single linkedlist functions in it.Then generate a shared object library file libslist.so from it.
	Implement below mentioned functions,
	SList *sl_create(void);
	int sl_isempty(SList *head);
	SList* sl_insert_first(SList *head, int ele);
	SList* sl_insert_last(SList *head, int ele);
	SList* sl_delete_first(SList *head, int *ele);
	SList* sl_delete_last(SList *head, int *ele);
	SList* sl_delete_element(SList *head, int *ele);
	SList* sl_insert_before(SList *head, int ele, int bele);
	SList* sl_insert_after(SList *head, int ele, int aele);
	SList* sl_deletelist(SList **head);
	void sl_printlist(SList **head);

( )	Create a library file named dlist.c and include all double linkedlist functions in it.Then generate a shared object library file libdlist.so from it.
	Implement below mentioned functions,
	DList *dl_create(void);
	int dl_isempty(DList *head);
	DList* dl_insert_first(DList *head, int ele);
	DList* dl_insert_last(DList *head, int ele);
	DList* dl_delete_first(DList *head, int *ele);
	DList* dl_delete_last(DList *head, int *ele);
	DList* dl_delete_element(DList *head, int *ele);
	DList* dl_insert_before(DList *head, int ele, int bele);
	DList* dl_insert_after(DList *head, int ele, int aele);
	DList* dl_deletelist(DList **head);
	void dl_printlist(DList **head);

( )	Write a Count() function that counts the number of times a given int occurs in a list.
( )	Write a GetNth() function that takes a linked list and an integer index and returns the data value stored in the node at that index position. GetNth() uses the C numbering convention.
( )	Write a function DeleteList() that takes a list, deallocates all of its memory and sets its head pointer to NULL (the empty list).

(Id) / Date	Assignment Topic
( ) _____	write a function InsertNth() which can insert a new node at any index within a list.
( ) _____	Write a SortedInsert() function which given a list that is sorted in increasing order, and a single node, inserts the node into the correct sorted position in the list.
( ) _____	Write an InsertSort() function which given a list, rearranges its nodes so they are sorted in increasing order. It should use SortedInsert().
( ) _____	Write an Append() function that takes two lists, 'a' and 'b', appends 'b' onto the end of 'a', and then sets 'b' to NULL (since it is now trailing off the end of 'a').
( ) _____	Given a list, split it into two sublists one for the front half, and one for the back half. If the number of elements is odd, the extra element should go in the front list.
( ) _____	Write a RemoveDuplicates() function which takes a list sorted in increasing order and deletes any duplicate nodes from the list. Ideally, the list should only be traversed once.
( ) _____	Write a SortedMerge() function that takes two lists, each of which is sorted in increasing order, and merges the two together into one list which is in increasing order.
( ) _____	Write an iterative Reverse() function that reverses a list by re-arranging all the .next pointers and the head pointer. Ideally, Reverse() should only need to make one pass of the list. The iterative solution is moderately complex.
( ) _____	

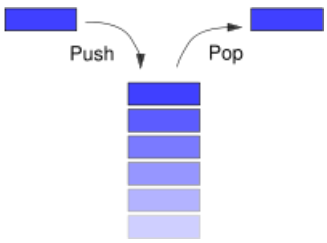
Chapter 3

Stack

3.1 Abstract

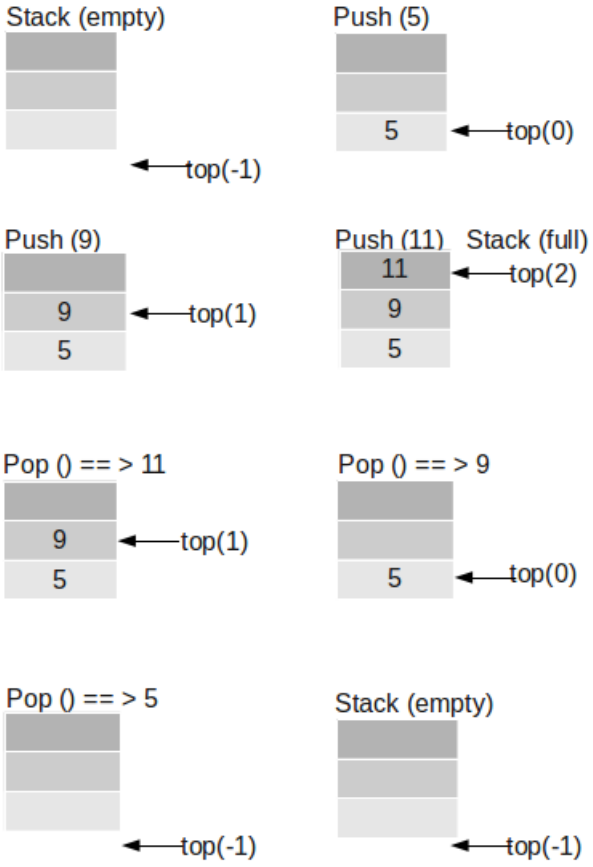
Stacks are ubiquitous in the computing world. Typicaly stack is a collection of items in which only the most recently added item may be removed. The latest added item is at the top. Basic operations are *push* and *pop*. Often *top* and *isEmpty* are available, too. Also known as *"last-in, first-out"* or *LIFO*.

Simply stact is a memory in which value are stored and retrieved in *"last in first out"* manner by using operations called *push* and *pop*.



3.2 Operations on a stack

Stack implemented with an array of capacity 3



The push operation adds a new item to the top of the stack, or initializes the stack if it is empty. If the stack is full and does not contain enough space to accept the given item, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items, or results in an empty stack, but if the stack is empty then it goes into underflow state (It means no items are present in stack to be removed).

- Create a new stack

- Add to the stack(Push)

- Delete from the stack(Pop)

- Check the next available(Top)

- Print the stack

- Destroy the stack

3.3 Example applications for stack

3.3.1 Converting a decimal number into a binary number

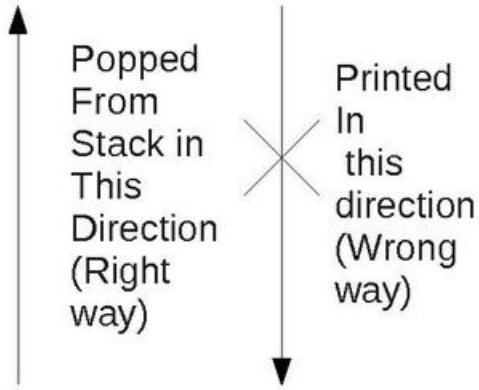
Algo:

```
Decimal to binary conversion of 23

 Read a number
 Iteration (while number is greater than zero)
 Find out the remainder after dividing the number by 2
 Print the remainder
 Divide the number by 2
 End the iteration
```

However, there is a problem with this logic. Suppose the number, whose binary form we want to find is 23. Using this logic, we get the result as 11101, instead of getting 10111.

To solve this problem, we use a stack. We make use of the LIFO property of the stack. Initially we push the binary digit formed into the stack, instead of printing it directly. After the entire number has been converted into the binary form, we pop one digit at a time from the stack and print it. Therefore we get the decimal number converted into its proper binary form.

2	23	
2	11	1
2	5	1
2	2	1
2	1	0
		1



3.3.2 Conversion of expressions

- Conversion from infix to postfix

Infix Expression : (((8 + 1) - (7 - 4)) / (11 - 9)

Postfix Expression : 8 1 + 7 4 - - 11 9 - /

Input	Operation	Stack (after op)	Output on monitor
(	(2.1) Push operand into stack	(	
(	(2.1) Push operand into stack	((	
(	(2.1) Push operand into stack	(((	
8	(2.2) Print it		8
+	(2.3) Push operator into stack	(((+	8
1	(2.2) Print it		8 1
)	(2.4) Pop from the stack: Since popped element is '+' print it	(((	8 1 +
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	((	8 1 +
-	(2.3) Push operator into stack	((-	
(	(2.1) Push operand into stack	((- (	
7	(2.2) Print it		8 1 + 7
-	(2.3) Push the operator in the stack	((- (-	
4	(2.2) Print it		8 1 + 7 4
)	(2.4) Pop from the stack: Since popped element is '-' print it	((- (	8 1 + 7 4 -
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	((-	
)	(2.4) Pop from the stack: Since popped element is '-' print it	((	8 1 + 7 4 - -
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	(	
/	(2.3) Push the operand into the stack	(/	
(	(2.1) Push into the stack	(/ (	
11	(2.2) Print it		8 1 + 7 4 - - 11
-	(2.3) Push the operand into the stack	(/ (-	
9	(2.2) Print it		8 1 + 7 4 - - 11 9
)	(2.4) Pop from the stack: Since popped element is '-' print it	(/ (	8 1 + 7 4 - - 11 9 -
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	(/	
)	(2.4) Pop from the stack: Since popped element is '/' print it	(	8 1 + 7 4 - - 11 9 - /
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	Stack is empty	
New line character	(2.5) STOP		

- Conversion from infix to prefix

3.3.3 Evaluation of expressions

- Evaluation of infix expression

Input String: (2 * 5 - 1 * 2) / (11 - 9)

Input Symbol	Character Stack (from bottom to top)	Integer Stack (from bottom to top)	Operation performed
(	(		
2	(	2	
*	(*		*Push* as * has higher priority
5	(*	2 5	
-	(*		Since '-' has less priority, we do 2 * 5 = 10
	(-	10	We push 10 and then push '-'
1	(-	10 1	
*	(- *	10 1	Push * as it has higher priority
2	(- *	10 1 2	
)	(-	10 2	Perform 1 * 2 = 2 and push it
	(	8	Pop - and 10 - 2 = 8 and push, Pop (
/	/	8	
(	/ (	8	
11	/ (	8 11	
-	/ (-	8 11	
9	/ (-	8 11 9	
)	/	8 2	Perform 11 - 9 = 2 and push it
New line		4	Perform 8 / 2 = 4 and push it
		4	Print the output, which is 4

- Evaluation of prefix expression

Input String: / - * 2 5 * 1 2 - 11 9

Input Symbol	Character Stack (from bottom to top)	Integer Stack (from bottom to top)	Operation performed
/	/		
-	/		
*	/ - *		
2	/ - * 2		
5	/ - * 2 5		
*	/ - * 2 5 *		
1	/ - * 2 5 * 1		
2	/ - * 2 5 * 1 2		
-	/ - * 2 5 * 1 2 -		
11	/ - * 2 5 * 1 2 - 11		
9	/ - * 2 5 * 1 2 - 11 9		
\n	/ - * 2 5 * 1 2 - 11	9	
	/ - * 2 5 * 1 2 -	9 11	
	/ - * 2 5 * 1 2	2	11 - 9 = 2
	/ - * 2 5 * 1	2 2	
	/ - * 2 5 *	2 2 1	
	/ - * 2 5	2 2	1 * 2 = 2
	/ - * 2	2 2 5	
	/ - *	2 2 5 2	
	/ -	2 2 10	5 * 2 = 10
	/	2 8	10 - 2 = 8
	Stack is empty	4	8 / 2 = 4
		Stack is empty	Print 4

- Evaluation of postfix expression

Infix Expression: 1 + 2 * 4 + 3
Postfix Expression: 1 2 4 * + 3 +

Input	Operation	Stack (after op)
1	Push operand	1
2	Push operand	2, 1
4	Push operand	4, 2, 1
*	Multiply	8, 1
+	Add	9
3	Push operand	3, 9
+	Add	12

3.4 Fuction call in C

3.4.1 Caller:

- Push parameters on the stack on reverse order (allows for varriable number of parameters).
- Push return address on stack.
- Jump to start of function.

3.4.2 Called function entry:

- Pushes local varriables on stack (just change stack pointer, no initilization).

3.4.3 Called function exit:

- place return value (if any) in register.
- Pop local varribles off stack.
- Jump to address at top of stack.

3.4.4 Caller:

- Pop return address and parameter off stack.

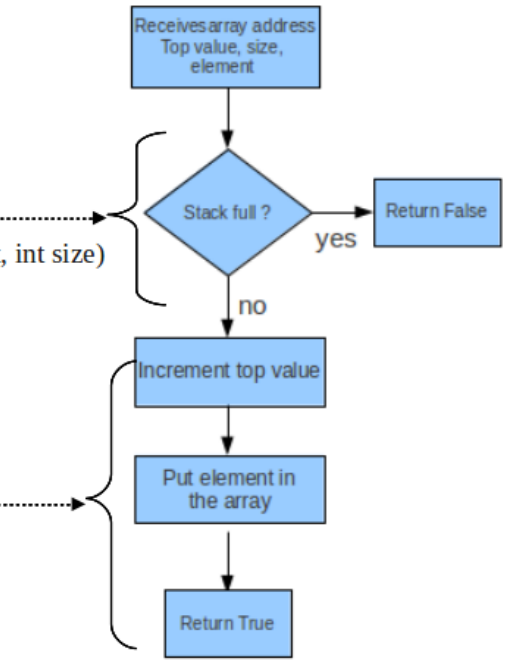
3.5 Lab Work

3.5.1 Practice:

StackPush_Array Function

1. Function passes the array pointer, top pointer, size of array & element to be inserted.
2. Check whether stack is full.
3. If yes, return False.
4. If no, increment top value.
5. Assign the element to the array pointed by top value.
6. Return True.

```
Int StackPush_Array ( int *array, int *top, int element, int size)
{
        if (isstackarrayfull (top, size) )
        {
                printf ("Stack Overflow");
                return FALSE;
        }

        else
        {       array [ ++(*top) ] = element;
                return TRUE;
        }
}
```



3.5.2 List of Assignments

(Id) / Date	Assignment Topic
( )	Create a library file named stack.c and include all stack functions in it.Then generate a shared object library file libstack.so from it.
( )	Do a stack program with most of the stack operation and use lib made by link list programs.
( )	Write programs to implement the following. Convert infix expressions to postfix expression. Convert infix expressions to prefix expression. Evaluate the infix expression using stack. Evaluate the prefix expression using stack. Evaluate the postfix expression using stack.

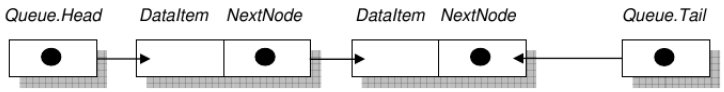
Chapter 4

Queue

4.1 Abstract

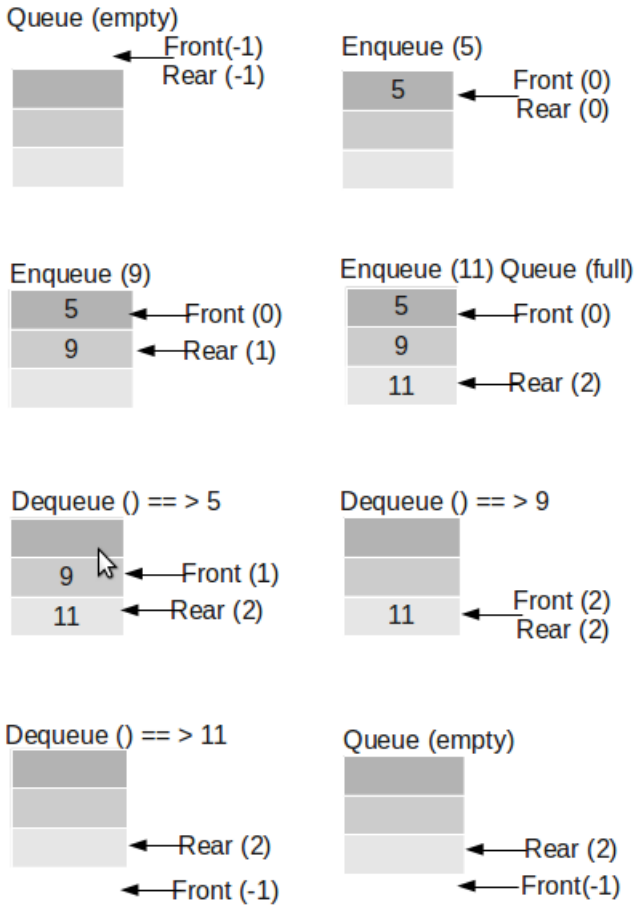
A collection of items in which only the earliest added item may be accessed. Basic operations are *add* or *enqueue* and *delete* or dequeue. *Delete* returns the item removed. Also known as *"first in first out"* or *FIFO*.

Queues occur naturally in situations where the rate at which clients demand for services can exceed the rate at which these services can be supplied. For example, in a network where many computers share only a few printers, the print jobs may accumulate in a print queue. In an operating system with a GUI, applications and windows communicate using messages, which are placed in message queues until they can be handled.



4.2 Operations on a queue

Queue implemented with an array of capacity 3



Queue overflow results from trying to add an element onto a full queue and queue underflow happens when trying to remove an element from an empty queue. Once the Rear reaches maximum capacity it cannot be incremented further. So the queue can not be used further until the rear and front are reset to minimum values.

- Create a new Queue

- Add to the queue(Enqueue)

- Delete from the queue (Dequeue)

- Print the queue

- Destroy the queue

4.3 Example application for queue

In general, queues are often used as "waiting lines". Here are a few examples of where queues would be used:

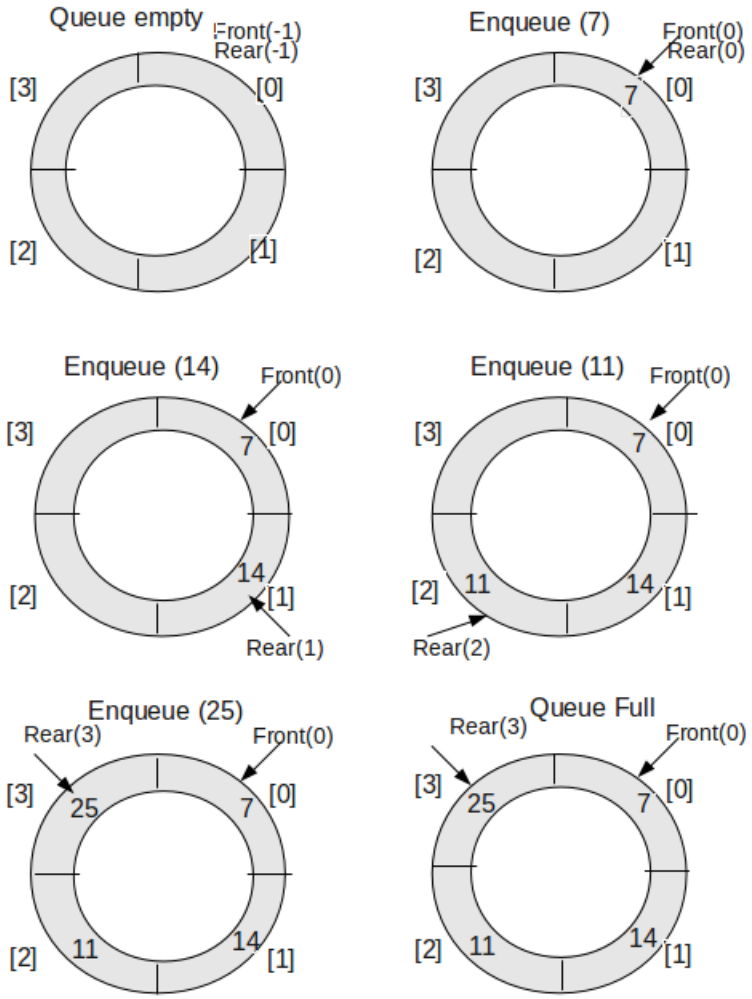
1. In operating systems, for controlling access to shared system resources such as printers, files, communication lines, disks and tapes. A specific example of print queues follows:

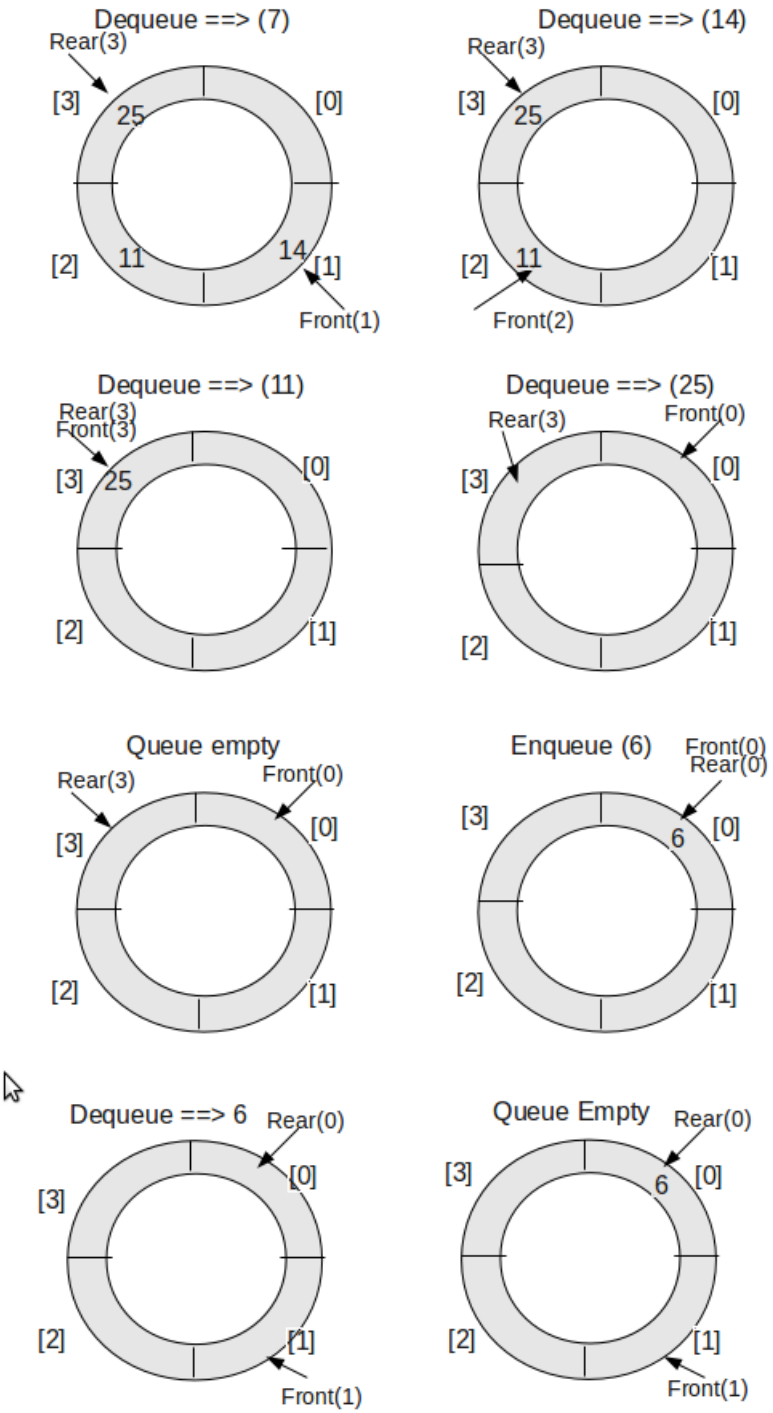
In the situation where there are multiple users or a networked computer system, you probably share a printer with other users. When you request to print a file, your request is added to the print queue. When your request reaches the front of the print queue, your file is printed. This ensures that only one person at a time has access to the printer and that this access is given on a first-come, first-served basis.

2. When placed on hold for telephone operators. For example, when you phone the toll-free number for your bank, you may get a recording that says, "Thank you for calling A-1 Bank. Your call will be answered by the next available operator. Please wait." This is a queuing system.

4.4 Circular Queues

Circular Queue inmplemented with an array of capacity 4





4.4.1 Difference in operations

- Create a new Queue
- Add to the queue(Enqueue)
- Delete from the queue (Dequeue)
- Print the queue
- Destroy the queue

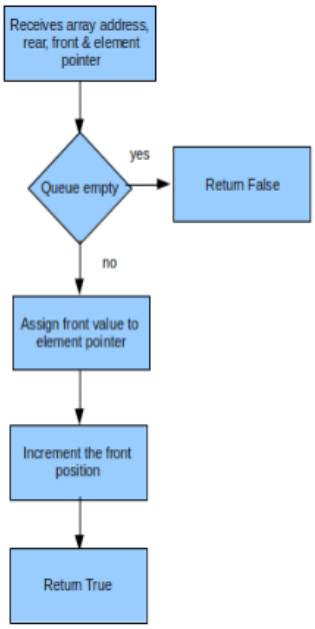
4.5 Lab Work

4.5.1 Practice:

Dequeue_Array Function

1. Function passes the array pointer, front pointer, rear & a variable to copy dequeued element.
2. Check whether queue is empty.
3. If yes return False.
4. If no, store the value in current front location to the element variable.
5. Increment the front position.
6. Return True.

```
Int Dequeue_Array ( int *array, int *front, int *rear, int element)
{
        if (isqueuearrayempty (*rear, *front) )
        {
                printf ("Queue Empty");
                return False;
        }

        else
        {       *element = array[(*front)++]
                return True;
        }
}
```



4.5.2 List of Assignments

(Id) / Date	Assignment Topic
( ) _____	Create a library file named queue.c and include all queue functions in it.Then generate a shared object library file libqueue.so from it.
( ) _____	Do a stack program with most of the queue operation and use lib made by link list programs.

Chapter 5

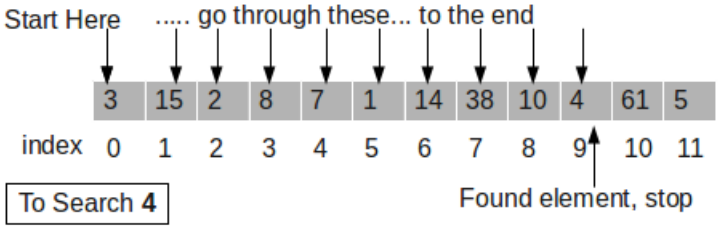
Day 4: Searching

5.1 Abstract

Search is to look for a value or item in a data structure. There are dozen of algorithms, data structures and approaches.

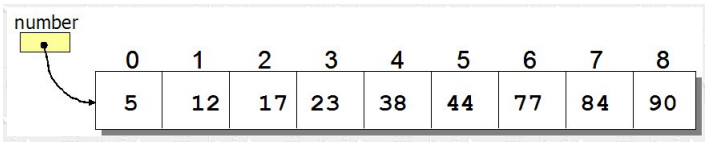
5.2 Linear Search

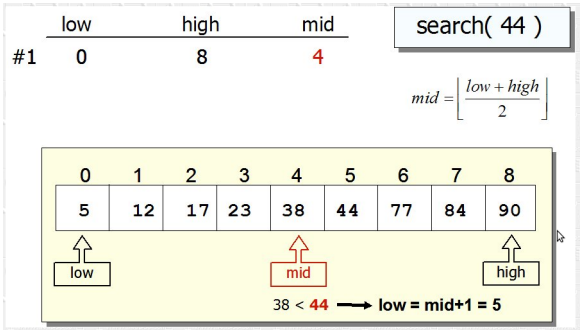
Search an array or list by checking items one at a time.

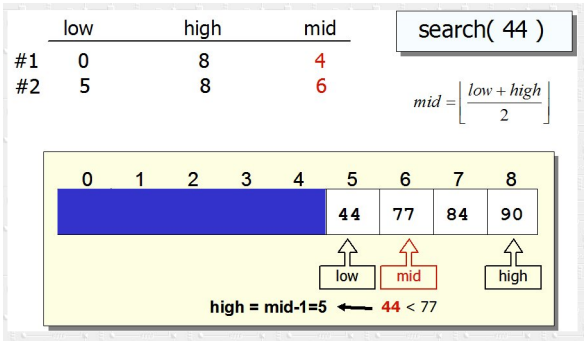


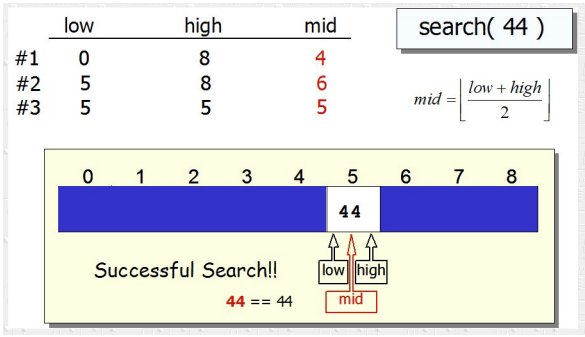
5.3 Binary Search

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. if the value of the search *key* is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check untill the value is found or interval is empty.









Algorithm: Recursive

```
BinarySearch(A[0..N-1], value, low, high) {
      if (high < low)
         return -1 // not found
      mid = (low + high) / 2
      if (A[mid] > value)
         return BinarySearch(A, value, low, mid-1)
      else if (A[mid] < value)
         return BinarySearch(A, value, mid+1, high)
      else
         return mid // found
 }
```

Algorithm: Iterative

```
      low = 0
      high = N
      while (low < high) {
         mid = (low + high)/2;
         if (A[mid] < value)
             low = mid + 1;
         else
              //can't be high = mid-1: here A[mid] >= value,
              //so high can't be < mid if A[mid] == value
              high = mid;
      }
      // high == low, using high or low depends on taste
      if ((low < N) && (A[low] == value))
         return low // found
      else
         return -1 // not found
```

5.4 Lab Work

5.4.1 List of Assignments

(Id) / Date	Assignment Topic
( ) _____	Implement all searching algorithms.
( ) _____	Implement Binary Searching using Recursion

Chapter 6

Day 5: Sorting

6.1 Abstract

Arrange items in a predetermined order. There are dozens of algorithms, the choice of which depends on factors such as the number of items relative to working memory, knowledge of the orderliness of the items or the range of the keys, the cost of comparing keys vs. the cost of moving items, etc. Most algorithms can be implemented as an in-place sort, and many can be implemented so they are stable, too.

6.2 Bubble Sort

Sort by comparing each adjacent pair of items in a list in turn, swapping the items if necessary, and repeating the pass through the list until no swaps are done.

Initial Arrangement	8	6	1	4	9	2	5	3	0
After first pass	6	1	4	8	2	5	3	0	9
After second pass	1	4	6	2	5	3	0	8	9
After third pass	1	4	2	5	3	0	6	8	9
After fourth pass	1	2	4	3	0	5	6	8	9
After tfifth pass	1	2	3	0	4	5	6	8	9
After sixth pass	1	2	0	3	4	5	6	8	9
After seventh pass	1	0	2	3	4	5	6	8	9
After eighth pass	0	1	2	3	4	5	6	8	9

Algo:

```
procedure bubbleSort( A : list of sortable items ) defined as:
   do
     swapped := false
     for each i in 0 to length( A ) - 1 do:
       if A[ i ] > A[ i + 1 ] then
          swap( A[ i ], A[ i + 1 ] )
          swapped := true
       end if
    end for
   while swapped
end procedure
```

Time complexity:

Best Case : $O(n)$
Average Case : $O(n^2)$
Worst Case : $O(n^2)$
Space complexity: 1

6.3 Insertion Sort

Sort by repeatedly taking the next item and inserting it into the final data structure in its proper order with respect to items already inserted. Run time is $O(n^2)$ because of moves.

Initial Arrangement	8	6	1	4	9	2	5	3	0
After first pass	6	8	1	4	9	2	5	3	0
After second pass	1	6	8	4	9	2	5	3	0
After third pass	1	4	6	8	9	2	5	3	0
After fourth pass	1	4	6	8	9	2	5	3	0
After tfifth pass	1	2	4	6	8	9	5	3	0
After sixth pass	1	2	4	5	6	8	9	3	0
After seventh pass	1	2	3	4	5	6	8	9	0
After eighth pass	0	1	2	3	4	5	6	8	9

Algo:

```
insertionSort(array A)
     for i = 1 to length[A]-1 do
     begin
          value = A[i]
          j = i-1
          while j >= 0 and A[j] > value do
          begin
          swap( A[j + 1], A[j] )
          j = j-1
          end
          A[j+1] = value
     end
```

Time complexity:
Best Case : $O(n)$
Average Case : $O(n^2)$
Worst Case : $O(n^2)$
Space complexity: 1

6.4 Selection Sort

A sort algorithm that repeatedly looks through remaining items to find the least one and moves it to its final location. The run time is $O(n^2)$, where n is the number of elements. The number of swaps is $O(n)$.

Initial Arrangement	8	6	1	4	9	2	5	3	0
After first pass	0	6	1	4	9	2	5	3	8
After second pass	0	1	6	4	9	2	5	3	8
After third pass	0	1	2	4	9	6	5	3	8
After fourth pass	0	1	2	3	9	6	5	4	8
After tfifth pass	0	1	2	3	4	6	5	9	8
After sixth pass	0	1	2	3	4	5	6	9	8
After seventh pass	0	1	2	3	4	5	6	8	9

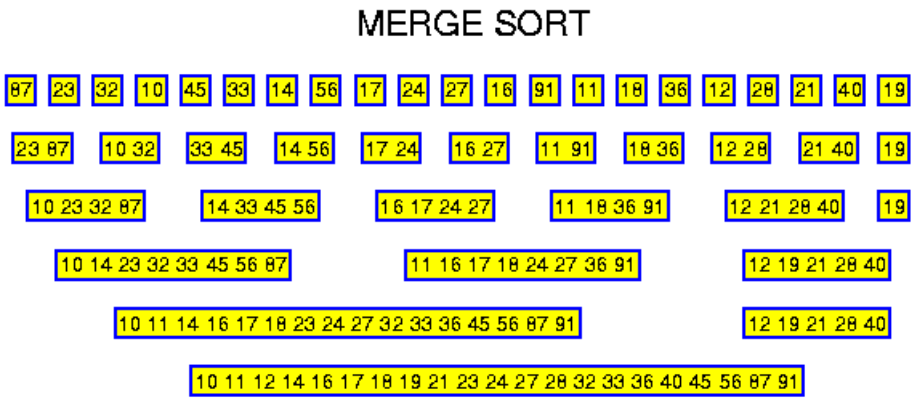
Algo:

```
for i ? 0 to n-2 do
     min ? i
     for j ? (i + 1) to n-1 do
         if A[j] < A[min]
             min ? j
     swap A[i] and A[min]
```

Time complexity:
Best Case : $O(n^2)$
Average Case : $O(n^2)$
Worst Case : $O(n^2)$
Space complexity: 1

6.5 Merge Sort

A sort algorithm that splits the items to be sorted into two groups, recursively sorts each group, and merges them into a final, sorted sequence. Run time is *O(n log n)*.



Algo:

```
function mergesort(m)
    var list left, right, result
    if length(m) ? 1
         return m
    var middle = length(m) / 2
    for each x in m up to middle
         add x to left
    for each x in m after middle
         add x to right
    left = mergesort(left)
    right = mergesort(right)
    result = merge(left, right)
    return result

function merge(left,right)
    var list result
    while length(left) > 0 and length(right) > 0
        if first(left) ? first(right)
            append first(left) to result
            left = rest(left)
        else
```

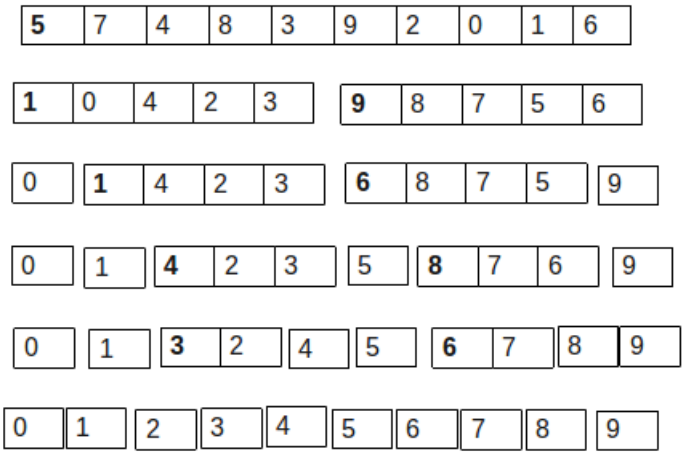
```
        append first(right) to result
        right = rest(right)
    end while
    if length(left) > 0
        append rest(left) to result
    if length(right) > 0
        append rest(right) to result
    return result
```

Time complexity:

Best Case : *O(n log n)*
Average Case : *O(n log n)*
Worst Case : *O(n log n)*
Space complexity: Depends; worst case is n

6.6 Quick Sort

Pick an element from the array (the pivot), partition the remaining elements into those greater than and less than this pivot, and recursively sort the partitions. There are many variants of the basic scheme above: to select the pivot, to partition the array, to stop the recursion on small partitions, etc.



*Pivot is highlightened in each array

Algo:

```
function partition(array, left, right, pivotIndex)
     pivotValue := array[pivotIndex]
     swap array[pivotIndex] and array[right] // Move pivot to end
     storeIndex := left
     for i  from  left to right ? 1
         if array[i] ? pivotValue
             swap array[i] and array[storeIndex]
             storeIndex := storeIndex + 1
     swap array[storeIndex] and array[right] // Move pivot to its final place
     return storeIndex

  procedure quicksort(array, left, right)
     if right > left
         select a pivot index (e.g. pivotIndex := left)
         pivotNewIndex := partition(array, left, right, pivotIndex)
         quicksort(array, left, pivotNewIndex - 1)
         quicksort(array, pivotNewIndex + 1, right)
```

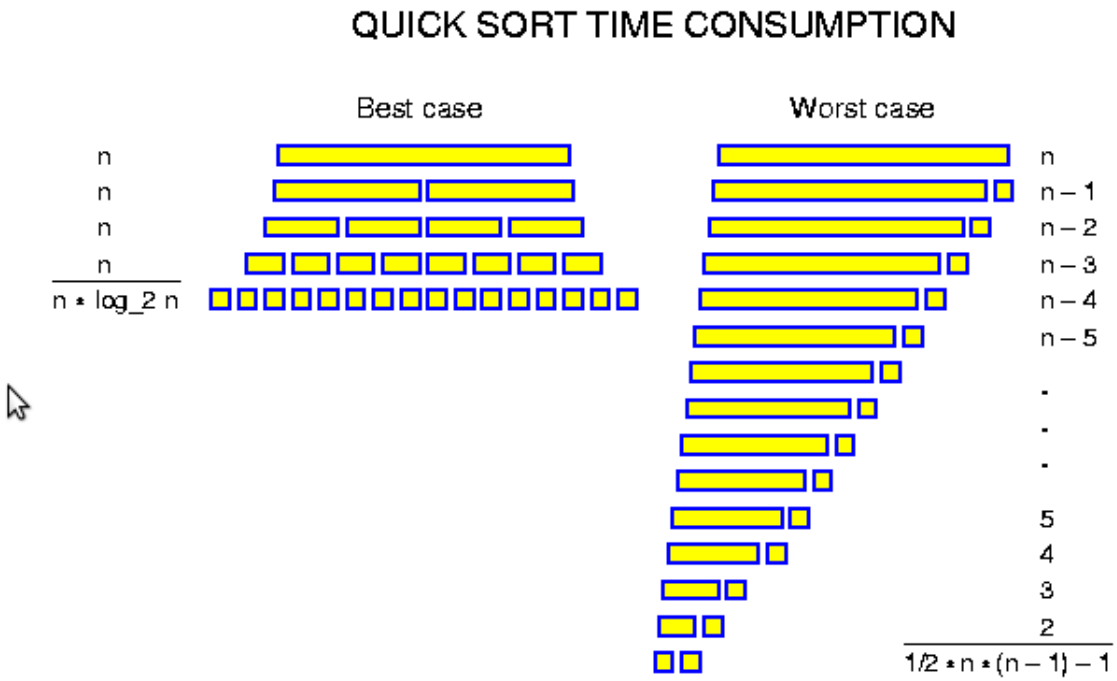
Time complexity:

Best Case : *O(n log n)*

Average Case : *O(n log n)*

Worst Case : $O(n^2)$

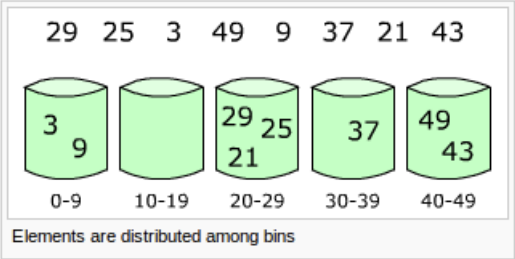
Space complexity: log n

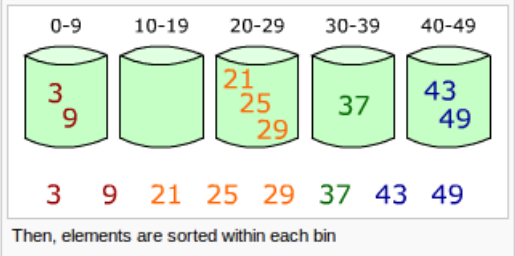


6.7 Other Sorts

6.7.1 Bucket Sort

A distribution sort where input elements are initially distributed to several buckets based on an interpolation of the element's key. Each bucket is sorted if necessary, and the buckets' contents are concatenated. Also known as bin sort.





Algo:

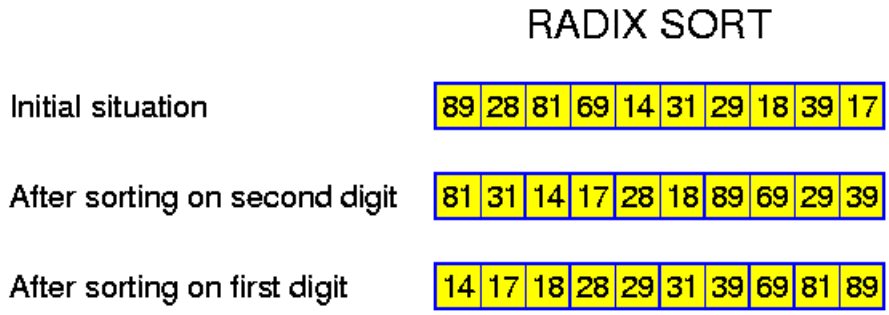
```
function bucket-sort(array, n) is
  buckets ? new array of n empty lists
  for i = 0 to (length(array)-1) do
    insert array[i] into buckets[msbits(array[i], k)]
  for i = 0 to n - 1 do
    next-sort(buckets[i])
  return the concatenation of buckets[0], ..., buckets[n-1]
```

Time complexity:

Space complexity:

6.7.2 Radix Sort

A multiple pass distribution sort algorithm that distributes each item to a bucket according to part of the item's key beginning with the least significant part of the key. After each pass, items are collected from the buckets, keeping the items in order, then redistributed according to the next most significant part of the key. A kind of distribution sort.



Time complexity:
Space complexity:

6.8 Lab Work

6.8.1 List of Assignments

(Id) / Date	Assignment Topic
( ) _____	Implement all sorting algorithms.
( ) _____	

Chapter 7

Trees

7.1 Abstract

A data structure accessed beginning at the root node. Each node is either a leaf or an internal node. An internal node has one or more child nodes and is called the parent of its child nodes. All children of the same node are siblings. Contrary to a physical tree, the root is usually depicted at the top of the structure, and the leaves are depicted at the bottom.

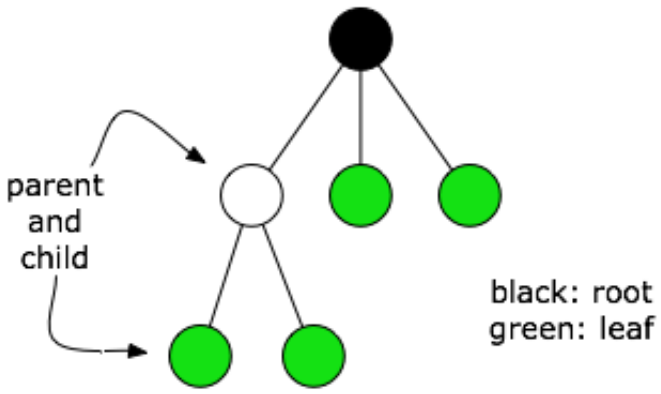
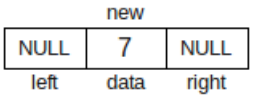


Figure: tree data structure

7.2 Operations on a Binary Search Tree

- Create a new node

  Drawing:

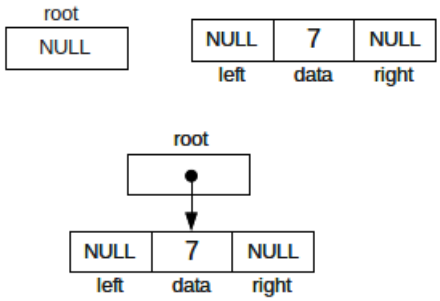


  Code:

```
struct _tree
{
        int data;
        struct _treet *left, *right;
};
typedef struct _tree Tree;

Tree *head = NULL;  //points to first node, now stores NULL

Tree* createnode (int element)
{
        Tree *new = NULL;

        new = (Tree*) malloc (sizeof (Tree)); //allocates node
        if (new == NULL)
        {
        //error , memory not allocated
        return;
        }
        new -> data = element;
        new -> left = NULL;
        new -> right = NULL;

return new;
}
```

- Insert into the tree
  - Insert a new node if tree is empty (root is NULL)

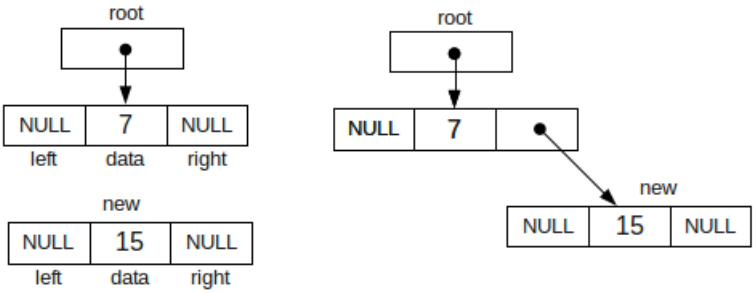
Insert node with data 7



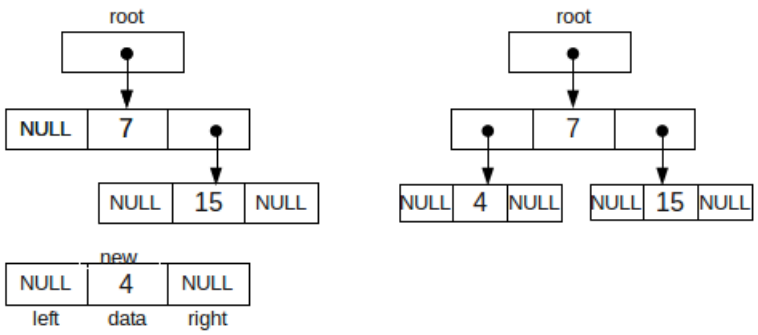
– Insert a new node if tree is non empty (root is not NULL)

  In order to insert a new node in the tree, its value is first compared with the value of the root. If its value is less than the root's, it is then compared with the value of the root's left child. If its value is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its value.

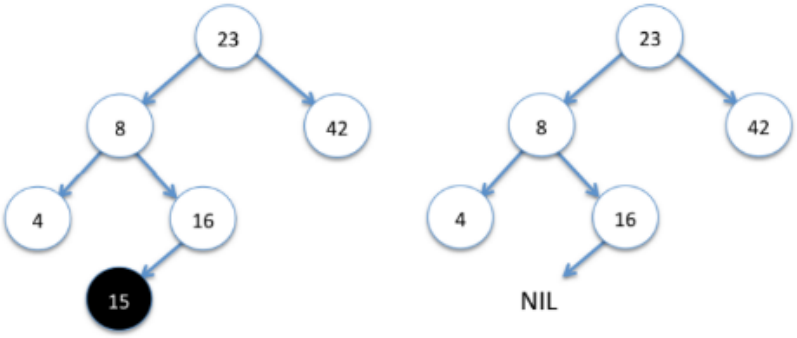
`Insert a node with data 15  (newnode->data > root->data)`



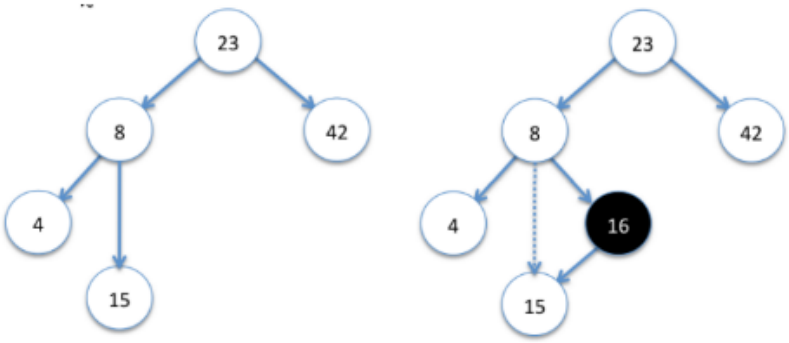
`Insert a node with data 4  (newnode->data < root->data)`



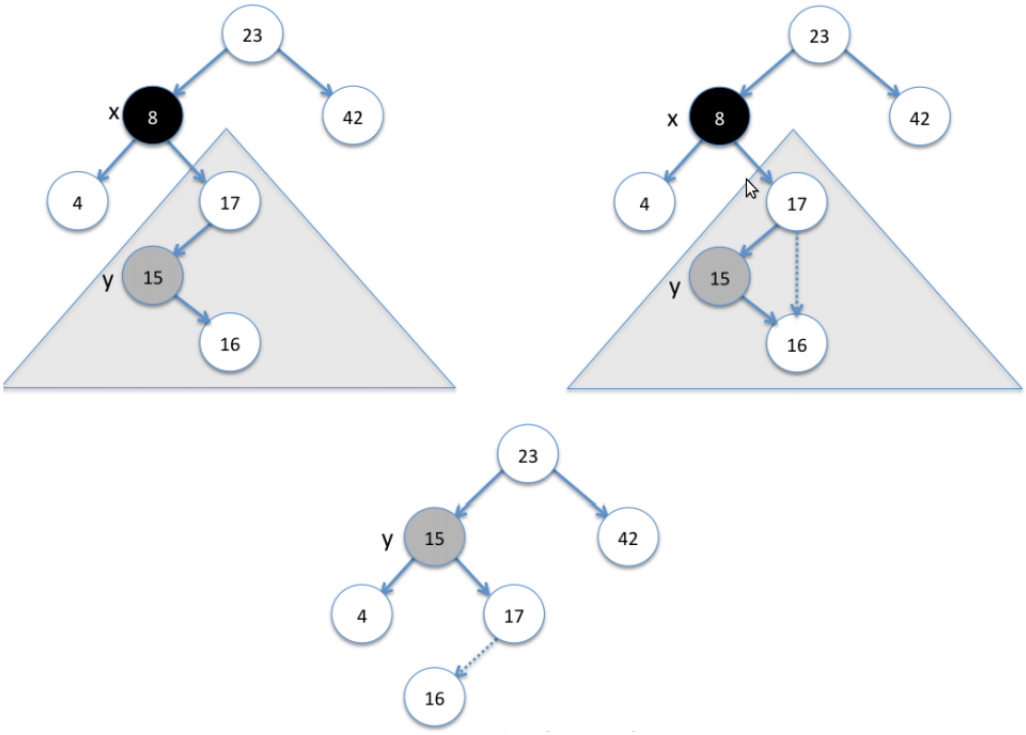
- Delete from the tree
  - Deleting a leaf



  - Deleting a node with one child



- Deleting a node with two children



- Traverse the tree
  - Inorder Traversal of the tree

    Algo:

```
if the tree is not empty
    traverse the left subtree
    visit the root
    traverse the right subtree
```

  - Preorder Traversal of the tree

    Algo:

```
if the tree is not empty
    visit the root
    traverse the left subtree
    traverse the right subtree
```

- Postorder Traversal of the tree
  Algo:

```
if the tree is not empty
    traverse the left subtree
    traverse the right subtree
    visit the root
```

- Search in the tree - BST

  A binary tree where every node's left subtree has keys less than the node's key, and every right subtree has keys greater than the node's key.

7.3 Applications

1. Storing a set of names, and being able to lookup based on a prefix of the name. (Used in internet routers.)

2. Storing a path in a graph, and being able to reverse any subsection of the path in O(log n) time. (Useful in travelling salesman problems).

7.3.1 Sorting - Heap Sort

A sort algorithm that builds a heap, then repeatedly extracts the maximum item. Run time is *O(n log n)*. A kind of *in-place* sort.

Algo:

```
function heapSort(a, count) is
     input:  an unordered array a of length count

     (first place a in max-heap order)
     heapify(a, count)
     end := count-1 //in languages with zero-based arrays the children are 2*i+1
     and 2*i+2
     while end > 0 do
           (swap the root(maximum value) of the heap with the last element of the
           heap)
           swap(a[end], a[0])
           (decrease the size of the heap by one so that the previous max value
           will stay in its proper placement)
           end := end - 1
           (put the heap back in max-heap order)
           siftDown(a, 0, end)


function heapify(a, count) is
     (start is assigned the index in a of the last parent node)
     start := (count - 2) / 2

     while start  0 do
           (sift down the node at index start to the proper place such that all
           nodes below the start index are in heap order)
   siftDown(a, start, count-1)
   start := start - 1
     (after sifting down the root all nodes/elements are in heap order)


function siftDown(a, start, end) is
     input:  end represents the limit of how far down the heap to sift.
     root := start

     while root * 2 + 1  end do    (While the root has at least one child)
```

```
child := root * 2 + 1        (root*2 + 1 points to the left child)
swap := root        (keeps track of child to swap with)
(check if root is smaller than left child)
if a[swap] < a[child]
        swap := child
(check if right child exists, and if it's bigger than what we're
currently swapping with)
if child+1  end and a[swap] < a[child+1]
       swap := child + 1
(check if we need to swap at all)
if swap != root
       swap(a[root], a[swap])
       root := swap     repeat to continue sifting down the child now)
else
      return
```

7.4 Lab Work

7.4.1 Practice

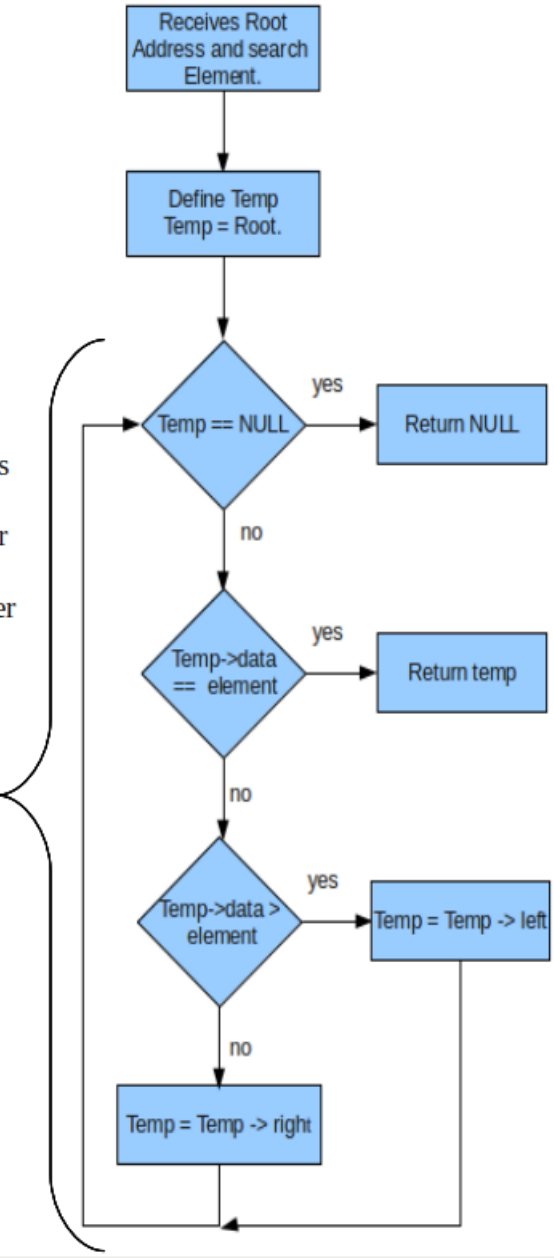
Write a function SearchNode that searches for a particular value in the tree.

SearchNode Function

Algorithm:

1. Function passes the root address and the element to be searched.
2. Define a temporary variable Temp;
3. Assign the Temp with root value.
4. Check whether the temp address is NULL.
5. If yes, return NULL.
6. If no, go to step 7
7. Check whether the data in the tempnode is equal to the search element.
8. If yes, return the temp address.
9. If no, go to step 10.
10. Check whether the data in the tempnode is greater than the search element.
11. If yes, assign temp with temp's left pointer address and goto step 4.
12. If no, assign temp with temp's right pointer address and goto step 4.

```
Tree* SearchNode ( Tree* root, int element )
{
        Tree *temp = root;

        while (temp != NULL )
        {
                if (temp -> data == element)
                        return temp;
                else if (temp -> data > element )
                        temp = temp -> left;
                else
                        temp = temp -> right;
        }
        return NULL;
}
```



7.4.2 List of Assignments

(Id) / Date	Assignment Topic
( ) _____	Create a library file named tree.c and include all tree functions in it.Then generate a shared object library file libtree.so from it. Implement below mentioned functions, Tree *bst_create(void); Tree *bst_insert(Tree *root, int element); Tree *bst_delete_node(Tree *root, int element); Tree *inorder_display(Tree *root); Tree *preorder_display(Tree *root); Tree *postorder_display(Tree *root); Tree *find_min(Tree *root); Tree *find_max(Tree *root); Tree *search_node(Tree *root, int element);
( ) _____	

Chapter 8

Hashing

8.1 Abstract

Hashing is a method to store data in an array so that storing, searching, inserting and deleting data is fast (in theory it's *O(1)*). For this every record needs an unique key.

The basic idea is not to search for the correct position of a record with comparisons but to compute the position within the array. The function that returns the position is called the 'hash function' and the array is called a 'hash table'.

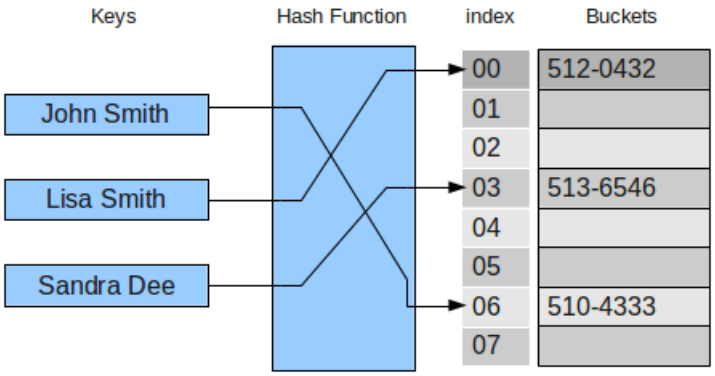
8.2 Hash function

A function that maps keys to integers, usually to get an even distribution on a smaller set of values.

A hash table or hash map is a data structure that uses a hash function to map identifying values, known as keys (e.g., a person's name), to their associated values (e.g., their telephone number). Thus, a hash table implements an associative array. The hash function is used to transform the key into the index (the hash) of an array element (the slot or bucket) where the corresponding value is to be sought.

In the below example the person's name is used as the key. Hash Function used is the sum of ascii value of the letters of the name modulus 10. When hash function is applied on the keys, the unique index is generated. Based on the index, their attributes (telephone numbers) are stored in the array.

```
For example :  John Smith ==> sum of ascii values % 10
                  = (74+111+104+110+83+109+105+116+104) % 10
                          = 6 (index)
```

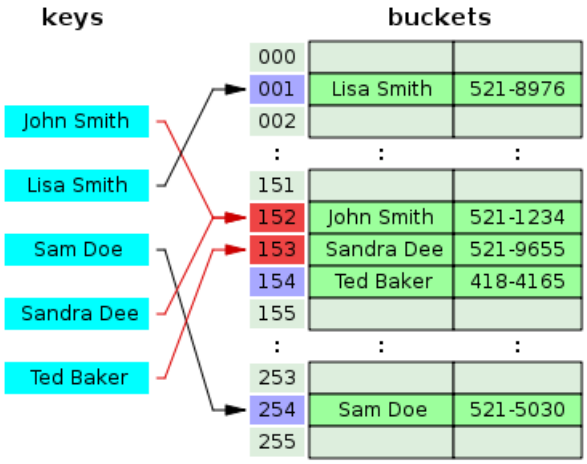


8.3 Collision handling

Ideally, the hash function should map each possible key to a unique slot index, but this ideal is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after it is created). Instead, most hash table designs assume that hash collisionsdifferent keys that map to the same hash valuewill occur and must be accommodated in some way.

8.4 Collision Handling Techniques

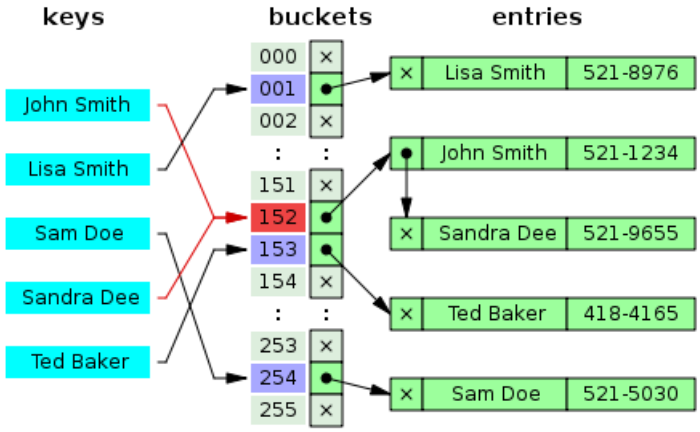
- Open Addressing



When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some probe sequence, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location ("address") of the item is not determined by its hash value.

Here when hash function is applied, John Smith got index 152. So John Smith is placed at index 152. Now for Sandra Dee also, the index is 152, which is not empty. So according to Open Addressing, Sandra is placed at index 153 which was empty. Now when Hash function is applied to Ted Baker, the index was 153, which is already filled. So it is placed at next empty slot which is 154.

- Separate Chaining



In the strategy known as separate chaining, direct chaining, or simply chaining, each slot of the bucket array is a pointer to a linked list that contains the key-value pairs that hashed to the same location. Lookup requires scanning the list for an entry with the given key. Insertion requires adding a new entry record to either end of the list belonging to the hashed slot. Deletion requires searching the list and removing the element.

Here When hash function applied John And Sandra got same index, 152. So from bucket 152 a linked list is started which connects all the keys with index 152.

8.5 Lab Work

8.5.1 List of Assignments

(Id) / Date	Assignment Topic
( ) _____	Create a database which stores the details of students : Name, age, sex and Parent's name. Implement it with Separate Chaining Hashing Technique.
( ) _____	

Appendix A

Assignment Guidelines

The following highlights common deficiencies which lead to loss of marks in Programming assignments. Review this sheet before turningin each Assignementt to make sure that the it is complete in all respects.

A.1 Quality of the Source Code

A.1.1 Variable Names

- Use variable names with a clear meaning in the context of the program whenever possible.

A.1.2 Indentation and Format

- Include adequate white-space in the program to improve readability. Insert blank lines to group sections of code. Use indentation to improve readability of control flow. Avoid confusing use of opening/closing braces.

A.1.3 Internal Comments

- Main program comments should describe overall purpose of the program. You should have a comment at the beginning of each source file describing what that file contains/does. Function comments should describe their purpose and other pertinent information, if any.

- Compound statements (control flow) should be commented. Finally, see that commenting is not overdone and redundant.

A.1.4 Modularity in Design

- Avoid accomplishing too many tasks in one function; use a separate module (Split your code into multiple logical functions). Also, avoid too many lines of code in a single module; create more modules. Design should facilitate individual module testing. Use automatic/local variables instead of external variables whenever possible. Use separate header files and implementation files for unrelated functions.

A.2 Program Performance

A.2.1 Correctness of Output

- Ensure that all outputs are correct. Incorrect outputs can lead to substantial loss in grade

A.2.2 Ease of Use

- The program should facilitate repeated use when used interactively and should allow easy exit. Requests for interactive input from the user should be clear. Incorrect user inputs should be captured and explained. Outputs should be well-formatted.

Appendix B

Grading of Programming Assignments

- Total points per assignment = 10
- Points for timely/early submission = 1
- The source code is out of 3 points. The distribution of points is as follows:
  - (a) The existence of the code itself (1 pts)
  - (b) Proper indentation of the code and comments (1 pts)
  - (c) Proper naming of the functions, variables + Modularity + (1 pts)
- You get 4 points if the program does exactly what it is supposed to do.
- Two (2) points are reserved for the ease of use, the type of user interface, the ability to handle various user input errors, or any extra features that your program might have.