Lab Manual

Data Structures With Files

S.E. Computer Sem III

Index

Sr. No.	Title of Programming Assignments	Page No.
1	BUBBLE SORT	3
2	SELECTION SORT	6
3	RECURSION	8
4	BINARY SEARCH	10
5	QUICK SORT	11
6	MERGE SORT	13
7	STACK	15
8	QUEUE	17
9	PRIORITY QUEUE	19
10	LINKED LIST	20
11	HASHING	26

BUBBLE SORT

AIM: Sort elements of an Array by using Bubble Sort

PLATFORM: JAVA

DESCRIPTION:

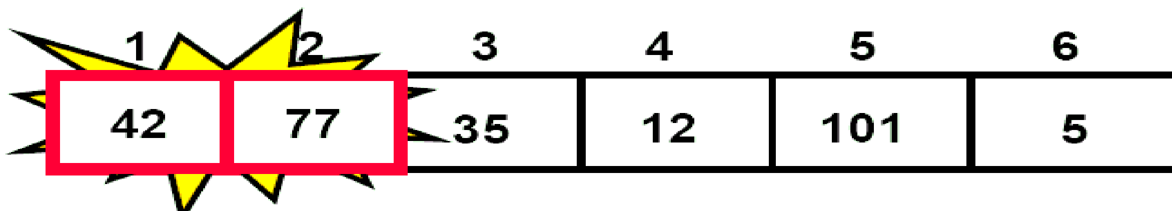
BUBBLE SORT:
Orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary

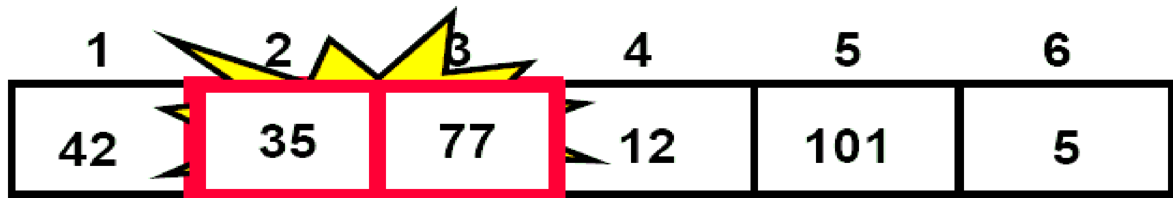
ALGORITHM:

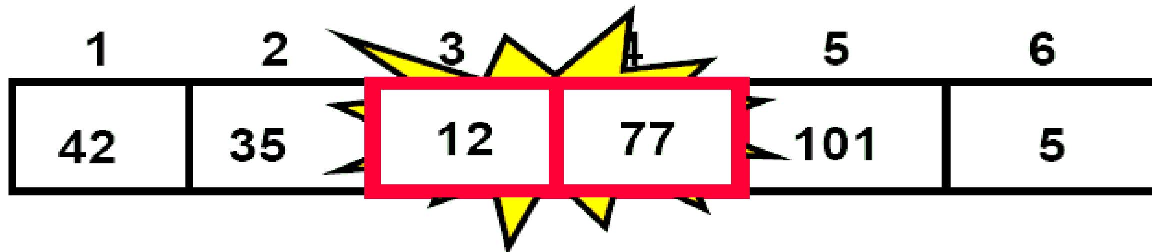
- scan the list, exchanging adjacent elements if they are not in relative order; this bubbles the highest value to the top
- scan the list again, bubbling up the second highest value
- repeat until all elements have been placed in their proper order

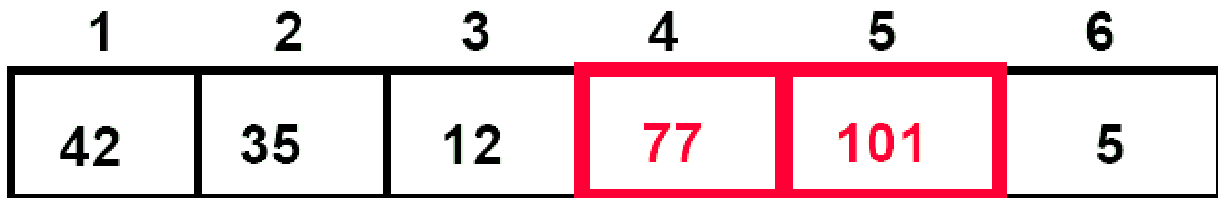
EXAMPLE:

- Traverse a collection of elements
- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping

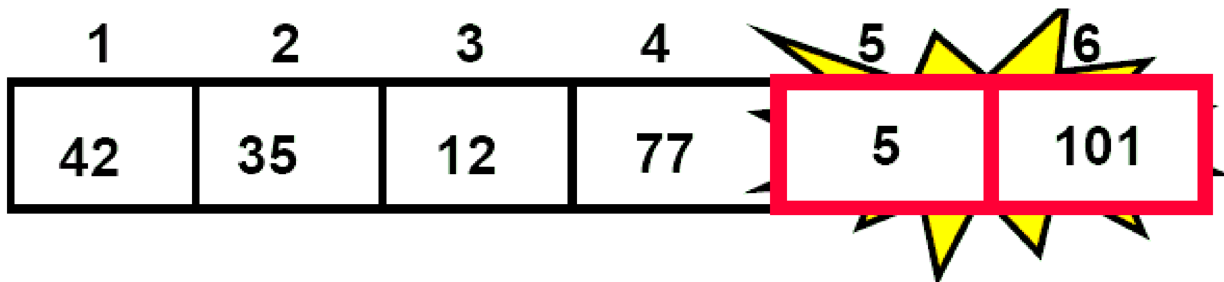








No need to swap



1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

BUBBLE SORT PSUEDO CODE:

```
public static void bubbleSort(int[] a) {
for (int i = 0; i < a.length; i++) {
for (int j = 1; j < a.length -i; j++) {
// swap adjacent out-of-order elements
if (a[j-1] > a[j]) {
swap(a, j-1, j);
}}
}}
```

SELECTION SORT

AIM: To Implement Sorting by using Selection Sort

PLATFORM: JAVA

PROBLEM STATEMENT: Sort elements of an Array by using Selection Sort

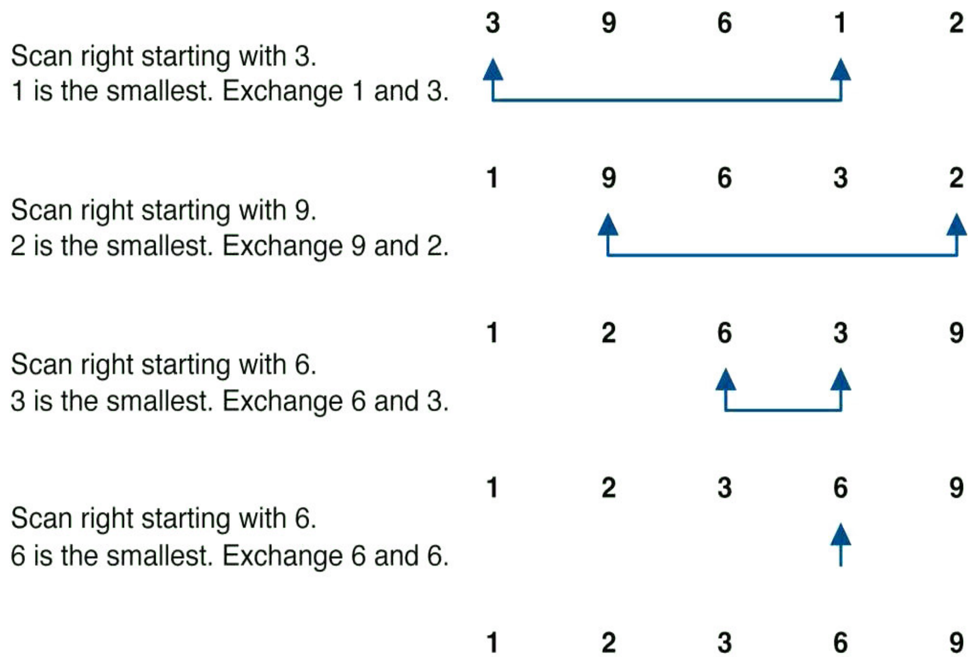
SELECTION SORT:

Orders a list of values by repetitively putting a particular value into its final position

ALGORITHM:

- find the smallest value in the list
- switch it with the value in the first position
- find the next smallest value in the list
- switch it with the value in the second position
- repeat until all values are in their proper places

EXAMPLE:



SELECTION SORT PSUDEO CODE:

```
public static void selectionSort(int[] a) {
for (int i = 0; i < a.length; i++) {
// find index of smallest element
int min = i;
for (int j = i + 1; j < a.length; j++) {
if (a[j] < a[min]) {
min = j;
}}/
/ swap smallest element with a[i]
swap(a, i, min);
}}
```

RECURSION

AIM: To Implement Tower of Hinoi and GCD by Using Recursion

PLATFORM: JAVA

DESCRIPTION:

A **recursive algorithm** is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input. More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem. For example, the elements of a recursively defined set, or the value of a recursively defined function can be obtained by a recursive algorithm.

If a set or a function is defined recursively, then a recursive algorithm to compute its members or values mirrors the definition. Initial steps of the recursive algorithm correspond to the basis clause of the recursive definition and they identify the basis elements. They are then followed by steps corresponding to the inductive clause, which reduce the computation for an element of one generation to that of elements of the immediately preceding generation.

In general, recursive computer programs require more memory and computation compared with iterative algorithms, but they are simpler and for many cases a natural way of thinking about the problem.

Example 1: Solve the Tower Of Hanoi Problem

Definition:

Given three posts (towers) and *n* disks of decreasing sizes, move the disks from one post to another one at a time without putting a larger disk on a smaller one. The minimum is 2^n-1 moves.

ALGORITHM

to move n disks from post A to post B

1) *recursively move the top n-1 disks from post A to C,*
2) *move the* n^{th} *disk from A to B, and*
3) *recursively move the n-1 disks from C to B.*

A solution using iteration *is: on odd-numbered moves, move the smallest disk clockwise. On even-numbered moves, make the single other move which is possible.*

Example 2: Decimal to Binary Conersion

```
DecToBin(int n)
{
        int r;
         if (n<=0)
         {
         cout<<"\n Binary Equavalent is :";
         return; }
         r = n % 2;
         dectobin(n/2);
         Print r;
    }
```

CONCLUSION: The Recursion has been implemented.

BINARY SEARCH

AIM: To Implement binary Search

PLATFORM: JAVA

DESCRIPTION:

a binary search is an algorithm for locating the position of an element in a sorted list. It inspects the middle element of the sorted list: if equal to the sought value, then the position has been found; otherwise, the upper half or lower half is chosen for further searching based on whether the sought value is greater than or less than the middle element. The method reduces the number of elements needed to be checked by a factor of two each time, and finds the sought value if it exists in the list or if not determines "not present", in logarithmic time.

ALGORITHM:

```
binary_search(array,value)
 first=0
last=array.size – 1
 while (first <= last)
mid = (first + last) / 2
    if (value > array[mid])
           first = mid + 1
    else if (value < array[mid])
           last = mid - 1
            else
             return true
    return false end
```

CONCLUSION: Binary Search has been Implemented

QUICK SORT

AIM: Sort elements of an Array by using Quick sort

PLATFORM: JAVA

DESCRIPTION:

QUICK SORT: orders a list of values by partitioning the list around one element called a pivot, then sorting each partition

ALGORITHM:

- choose one element in the list to be the pivot (partition element)
- organize the elements so that all elements less than the pivot are to its left and all greater are to its right
- apply the quick sort algorithm (recursively) to both partitions

Divide the work into two methods:

- quickSort – performs the recursive algorithm
- partition – rearranges the elements into two partitions

QUICK SORT PSEUDO-CODE

1. Let S be the input set.
2. If |S| = 0or |S| = 1, then return.
3. Pick an element vin S. Call vthe pivot.

4. Partition S –{v}into two disjoint groups:

    S1= {x S –{v}| x v}

    S2= {x S –{v}| x v}

 Return { quicksort(S_1), v, quicksort(S_2) }

Partitioning algorithm

1. Move the pivotto the rightmostposition.
2. Starting from the left, find an element pivot. Call the position i.
3. Starting from the right, find an element pivot. Call the position j.
4. Swap S[i]and S[j].

8	1	4	9	0	3	5	2	7	6

0 9

OUTPUT :

0 1 2 3 4 5 6 7 8 9

POST LAB ASSIGNMENT:

Quick Sort by using middle element as a pivot element

Quick Sort by using any element as a pivot element

MERGE SORT

AIM: Sort elements of an Array by using Merge Sort .

PLATFORM: JAVA

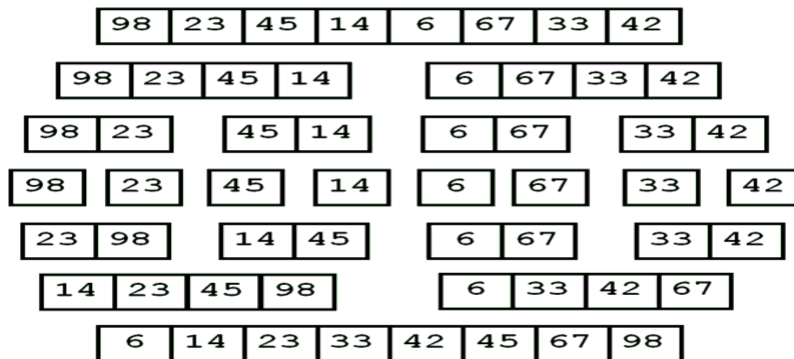
MERGE SORT:

Orders a list of values by recursively dividing the list in half until each sub-list has one element, then recombining

ALGORITHM:

- divide the list into two roughly equal parts
- recursively divide each part in half, continuing until a part contains only one element
- merge the two parts into one sorted list

EXAMPLE



Merging two sorted arrays

1. **Merge Operation:**

- Given two sorted arrays, merge operation produces a sorted array with all the elements of the two arrays

A

6	13	18	21

B

4	8	9	20

C

4	6	8	9	13	18	20	21

Running time of merge: O(n), where n is the number of elements in the merged array.

when merging two sorted parts of the same array, we'll need a temporary array to store the merged whole.

CONCLUSION: Merge Sort has been Implemented

STACK

AIM: To Implement Stack Using Array

PLATFORM: JAVA

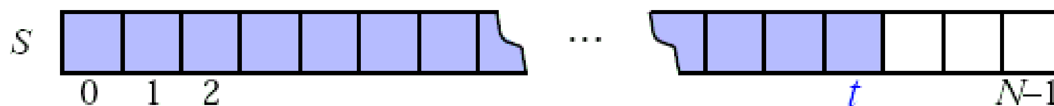
DESCRIPTION:

Stacks

- A stack is a container of objects that are inserted and removed according to the last-in-first-out (LIFO) principle.
- Inserting an item is known as “pushing” onto the stack. “Popping” off the stack is synonymous with removing an item.

An Array-Based Stack Algorithm:

- Create a stack using an array by specifying a maximum size N for our stack, e.g. N = 1,024.
- The stack consists of an N-element array S and an integer variable t, the index of the top element in array S.



- Array indices start at 0, so we initialize t to -1

PSEUDO-CODE

Algorithm size():

return t +1

Algorithm isEmpty():

return (t < 0)

Algorithm top():

if isEmpty() then throw a StackEmptyException

return S[t]

Algorithm push(o):

if size() = N then throw a StackFullException

t = t + 1

S[t] o

Algorithm pop():

if isEmpty() then

throw a StackEmptyException

e S[t]

S[t] null

t t-1

return e

PERFORMANCE:

- Let n be the number of elements in the stack the space used is O(n)
- Each operation runs in time O(1)
- The array implementation is simple and efficient.

LIMITATIONS:

- The maximum size of the stack must be defined a priori and cannot be changed
- Trying to push a new element into a full stack causes an implementation-specific exception
- There is an upper bound, N, on the size of the stack. The arbitrary value N may be too small for a given application, or a waste of memory.

CONCLUSION:

The stack has been implemented.

QUEUE

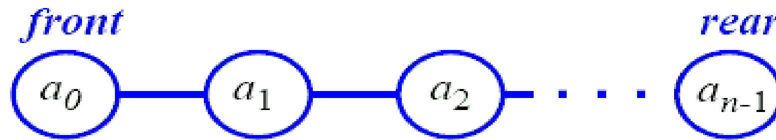
AIM: To Implement Queue Using Array

PLATFORM: JAVA

DESCRIPTION:

Queues

- A queue is a container of objects that are inserted and removed according to first-in-first-out (FIFO) principle.
- Elements are inserted at the rear (enqueued) and removed from the front (dequeued)

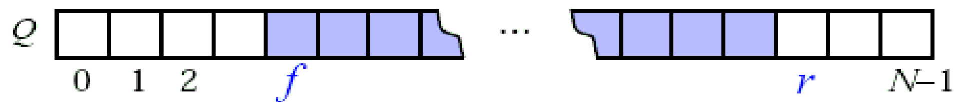


An Array-Based Queue Algorithm

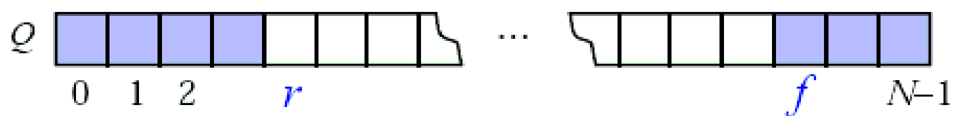
- Create a queue using an array in a circular fashion
- A maximum size N is specified, e.g. N = 1,000.
- The queue consists of an N-element array Q and two integer variables:

  f, index of the front element

  r, index of the element after the rear one
- "normal configuration"



- "wrapped around" configuration



Pseudo-Code

Algorithm size():

return (N - f + r) mod N

Algorithm isEmpty():

return (f = r)

Algorithm front():

if isEmpty() then throw a QueueEmptyException

return Q[f]

Algorithm dequeue():

if isEmpty() then throw a QueueEmptyException

temp Q[f]

Q[f]   null

f (f + 1) mod N

return temp

Algorithm enqueue(o):

if size = N - 1 then throw a QueueFullException

Q[r] o

PERFORMANCE:

- Let n be the number of elements in the stack the space used is O(n)
- Each operation runs in time O(1)
- The array implementation is simple and efficient.

LIMITATIONS:

- The maximum size of the stack must be defined a priori and cannot be changed
- Trying to push a new element into a full stack causes an implementation-specific exception
- There is an upper bound, N, on the size of the stack. The arbitrary value N may be too small for a given application, or a waste of memory.

CONCLUSION: The queue has been implemented.

PRIORITY QUEUE

AIM: To Implement Priority Queue

PLATFORM: JAVA

DESCRIPTION:

A priority queue is an abstract data type that supports the following three operations:

- InsertWithPriority: add an element to the queue with an associated priority

- GetNext: remove the element from the queue that has the highest priority, and return it (also known as "PopElement(Off)", or "GetMinimum")
- PeekAtNext (optional): look at the element with highest priority without removing it

IMPLEMENTATIONS:

1. list: store all customers/jobs in an unordered list, remove min/max one by searching for it problem: expensive to search
2. sorted list: store all in a sorted list, then search it in O(log n) time with binary search problem expensive to add/remove
3. binary search tree: store in a BST, search it in O(log n) time for the min (leftmost) element problem: tree could be unbalanced on nonrandom input
4. balanced BST

   a. problem: in practice, if the program has many adds/removes, it performs slowly on AVL trees and other balanced BSTs

   b. problem: removal always occurs from the left side, which

CONCLUSION: The Priority Queue has been implemented.

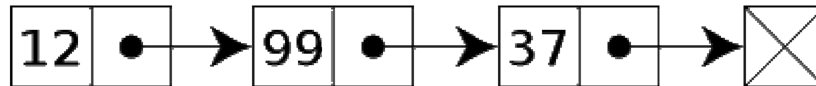
LINKED LIST

AIM: To Implement All types Of Linked List

PLATFORM: JAVA

DESCRIPTION:

`linked list` is a data structure that consists of a sequence of data records such that in each record there is a field that contains a reference (i.e., *`link`*) to the next record in the sequence.



A linked list whose nodes contain two fields: an integer value and a link to the next node

Linked lists are among the simplest and most common data structures, and are used to implement many important abstract data structures, such as stacks, queues, hash tables, symbolic expressions, skip lists, and many more.

The principal benefit of a linked list over a conventional array is that the order of the linked items may be different from the order that the data items are stored in memory or on disk. For that reason, linked lists allow insertion and removal of nodes at any point in the list, with a constant number of operations.

SINGLY-LINKED LIST

AIM: To Implement Singly-Linked List

PLATFORM: JAVA

DESCRIPTION:

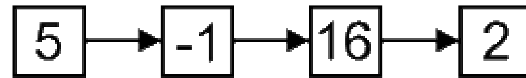
SINGLY-LINKED LIST

Linked list is a very important dynamic data structure. Basically, there are two types of linked list, singly-linked list and doubly-linked list. In a singly-linked list every element contains some data and a link to the next element, which allows to keep the structure. On the other hand, every node in a doubly-linked list also contains a link to the previous node. Linked list can be an underlying data structure to

implement stack, queue or sorted list.

Example

Sketchy, singly-linked list can be shown like this:



Each cell is called a **node** of a singly-linked list. First node is called **head** and it's a dedicated node. By knowing it, we can access every other node in the list. Sometimes, last node, called **tail**, is also stored in order to speed up add operation.

Operations on a singly-linked list

Concrete implementation of operations on the singly-linked list depends on the purpose, it is used for. Following the links below, you can find descriptions of the common concepts, proper for every implementation.

- Singly-linked list traversal
- Adding a node
- Removing a node

Traversal algorithm

Beginning from the head,

1. check, if the end of a list hasn't been reached yet;
2. do some actions with the current node, which is specific for particular algorithm;
3. current node becomes previous and next node becomes current. Go to the step 1.

Adding A Node

Insertion into a singly-linked list has two special cases. It's insertion a new node before the head (to the very beginning of the list) and after the tail (to the very end of the list). In any other case, new node is inserted in the middle of the list and so, has a predecessor and successor in the list. There is a description of all these cases below.

Empty list case

When list is empty, which is indicated by (head == NULL)condition, the insertion is quite simple. Algorithm sets both head and tail to point to the new node.

Add first

In this case, new node is inserted right before the current head node.

It can be done in two steps:

1. Update the next link of a new node, to point to the current head node.
2. Update head link to point to the new node.

Add last

In this case, new node is inserted right after the current tail node.

It can be done in two steps:

1. Update the next link of the current tail node, to point to the new node.
2. Update tail link to point to the new node.

General case

In general case, new node is **always inserted between** two nodes, which are already in the list. Head and tail links are not updated in this case.

Such an insert can be done in two steps:

1. Update link of the "previous" node, to point to the new node.
2. Update link of the new node, to point to the "next" node.

DOUBLY-LINKED LIST

AIM: To Implement Doubly-Linked List

PLATFORM: JAVA

DESCRIPTION:

DOUBLY-LINKED LIST

Doubly-linked List is a linked data structure that consists of a set of data records, each having two

special *link* fields that contain references to the previous and to the next record in the sequence. It can be viewed as two singly-linked lists formed from the same data items, in two opposite orders.

ALGORITHMS

1. Open doubly-linked lists

Data type declarations

```
record Node {
data // The data being stored in the node
next // A reference to the next node; null for last node
prev // A reference to the previous node; null for first node
 }

record List {
Node firstNode   // points to first node of list; null for empty list
Node lastNode    // points to last node of list; null for empty list
}
```

Iterating over the nodes

Iterating through a doubly linked list can be done in either direction. In fact, direction can change many times, if desired.

Forwards

```
node := list.firstNode
while node ≠ null
    <do something with node.data>
    node := node.next
```

Backwards

```
node := list.lastNode
while node ≠ null
    <do something with node.data>
    node := node.prev
```

2. Inserting a node

These symmetric functions add a node either after or before a given node, with the diagram demonstrating after:

Insert After

```
insertAfter(List list, Node node, Node newNode)
newNode.prev := node
newNode.next := node.next
```

```
if node.next == null
list.lastNode := newNode
else
node.next.prev := newNode
node.next := newNode
```

Insert Before

```
insertBefore(List list, Node node, Node newNode)
newNode.prev := node.prev
newNode.next := node
if node.prev is null
list.firstNode := newNode
else
node.prev.next := newNode
node.prev    := newNode
```

3. Deleting a node

Removing a node is easier, only requiring care with the *firstNode* and *lastNode*:

Remove

```
        remove(List list, Node node)
        if node.prev == null
list.firstNode := node.next
        else
node.prev.next := node.next
        if node.next == null
list.lastNode := node.prev
        else
node.next.prev := node.prev
        destroy node
```

HASHING

AIM: To Implement Hash function

PLATFORM: JAVA

DESCRIPTION:

hashing

To store data into the hash table, one must know that in order to do this, a certain value must be divided. For example if the data is consist of n integers and we have k number of cells. Then the address wherein the data would be stored is the function:
hash= n % k;
The above algorithm will only work if the given data is in integer. How about strings or character datas? For strings of integers this is done by splitting the string into equal numbers of substring, where each substring can be represented as an integer. In this way, the n in the hash function can be found by adding the value of each substring. While in the character scenario, this can be solved by assigning each letter to their corresponding place in the alphabet. For example, the letter A is 1 while Z is 26. In this manner, all letters has their own number correspondence. In the string of character problem, this problem can be solved by just adding the corresponding number of each letter, thus gaining a single number to be hashed.
The main problem that would occur is the collision, this is a scenario wherein, there will be a limited number of memory, and two or more data can collide in a single address. This problem can be solved by a rehashing algorithm
.

Linear Probing

The need to have a rehash function arises when a collision occurs. This happens when two or more information would collide on the same cell allocated for the hash table. Thus, a rehashing is needed. Since we know that the hash function for storing the information or data into the allocated memory is done by getting the remainder of the data's numerical equivalent divide by the number of space allocated. Then, the rehashing function is the method for finding the second or third or so on location for the information.
One rehashing technique is the Linear Probing, where the rehashing is done by looking for the next empty space that it can occupy. The function for the rehashing is the following:
rehash(key)=(n+1) % k;
This method works in such a way that if the first location is not free, then it will go to the next location and check if that location is free or not, and so on until it finds a free location or can't find anyone at all.
For formality and familiarity's sake, an empty space would be given a -1 value while a deleted data's space would be -2. In this way, finding an empty space is easy and also the search for a stored item would be easier.

ALGORITHM
INSERT

```
  void insert( key, r )
    typekey key;  dataarray r;

    { extern int n;
      int i, last;

    i = hashfunction( key ) ;
    last = (i+m-1) % m;
    while ( i!=last && !empty(r[i]) && !deleted(r[i]) && r[i].k!=key )
         i = (i+1) % m;
    if (empty(r[i]) || deleted(r[i]))
            {
            /*** insert here ***/
            r[i].k = key;
            n++;
            }
    else Error     /*** table full, or key already in table ***/;
    }
```

SEARCH

```
  int search( key, r )
    typekey key;  dataarray r;

    { int i, last;
```

```
i = hashfunction( key ) ;
last = (i+n-1) % m;
while ( i!=last && !empty(r[i]) && r[i].k!=key )
    i = (i+1) % m;
if (r[i].k==key)  return( i );
    else        return( -1 );
}
```

EXAMPLE(Linear Probing)
For example, 5 spaces for integers. Input: 1,5,27,25 21, 30. Delete 5. Look for 25, 29.
Solution:
INPUT:
*To insert 1, 1 %5=1, therefore 1 is stored at array[1].
*To insert 5, 5%5=0, therefore 5 is stored at array[0].
*To insert 27, 27%5=2, therefore off to array[2].
*To insert 25, 25 % 5=0, since 0 is occupied, rehashing is done, 26%5=1, rehashing is done again, 27%5=2, then 28%5=3. Since 3 is -1, 25 can be stored here.
*30 cannot b e stored for insufficient number of memory allocation.
DELETE:
*Look for 5, using the hash, we found 5 at location 0. Then, we assign a new value -2 location 0 to indicate this value have been deleted.
RETRIEVE:

- To look for 25, we use the hash key, to find its location. And found 0, but then the value stored I n array[0] is not the same, therefore, the rehashing would be used. Then we traverse the hash table, until we found the right place.
- To see if 29 does exist, we use the hash and rehash to find if they exist.

QUADRATIC PROBING

Quadratic probing is a scheme in computer programming for resolving collisions in hash tables.

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This algorithm is used in open-addressed hash tables. Quadratic probing provides good memory caching because it preserves some locality of reference; however, linear probing has greater locality and, thus, better cache performance. Quadratic probing better avoids the clustering problem that can occur with linear probing, although it is not immune.

Quadratic probing is used in the Berkeley Fast File System to allocate free blocks. The allocation routine chooses a new cylinder-group when the current is nearly full using quadratic probing, because of the speed it shows in finding unused cylinder-groups.

ALGORITHM

1. Let $h(k)$ be a hash function that maps an element k to an integer in $[0,m-1]$, where m is the size of the table.
2. Let the ith probe position for a value k be given by the function $h(k,i) = (h(k) + c_1 i + c_2 i^2)(\mathrm{mod}\ m)$, where C_2 !=0.
3. If $c_2 = 0$, then $h(k,i)$ degrades to a linear probe. For a given hash table, the values of c_1 and c_2 remain constant.

Example: If $h(k,i) = (h(k) + i + i^2)(\mathrm{mod}\ m)$, then the probe sequence will be $h(k), h(k) + 2, h(k) + 6, \ldots$

For $m = 2^n$, a good choice for the constants are $c_1 = c_2 = 1/2$, as the values $h(k,i)$ for i in $[0,m-1]$ are all distinct.This leads to a probe sequence of $h(k), h(k) + 1, h(k) + 3, h(k) + 6, \ldots$ where the values increase by 1,2,3,....

For prime $m > 2$, most choices of c_1 and c_2 will make $h(k,i)$ distinct for i in $[0,(m-1)/2]$. Such choices include $c_1 = c_2 = 1/2$, $c_1 = c_2 = 1$, and $c_1 = 0, c_2 = 1$. Because there are only about m/2 distinct probes for a given element, it is difficult to guarantee that insertions will succeed when the load factor is > 1/2.

CONCLUSION: The Hashing has been implemented.

Depth First Search:-

Here, the starting point can be any vertex and the next vertex to be visited is decided by the traversal you choose. If you choose the DFS, the next vertex to be visited will be the adjacent vertex (to the starting point), which has the highest depth value(??????...look at the graph below) and then the next adjacent vertex with the next higher depth value and so on till all the adjacent nodes for that vertex are not visited. We repeat this same procedure for every visited vertex of the graph.

Well, I got to explain this with the help of a graph -

If V1 is the starting point of my traversal then, as you can see - V2, V3 and V8 are its adjacent vertices and so in the Adjacency List of V1. Now, I am suppose to visit these vertices, but as this is DFS, I should visit V8 first and then V2 and V3. So, to keep track of which vertex to be visited next, we make use of a STACK, which holds these vertex values in the DFS order. Let's see the algorithm...

Algorithm:-

Step-1: Set the starting point of traversal and push it inside the stack

Step-2: Pop the stack and add the popped vertex to the list of visited vertices

Step-3: Find the adjacent vertices for the most recently visited vertex(from the Adjacency Matrix)

Step-4: Push these adjacent vertices inside the stack (in the increasing order of their depth) if they are not visited and not there in the stack already

Step-5: Go to step-2 if the stack is not empty