Subject:-Data Structures using C
Subject code: - BCA 108

UNIT-I

Introduction to Data Structures : Basic Terminology, Elementary Data Organizations, Classification of data structures and its operations.

Arrays: Representation of single and multidimensional arrays (up to three dimensions) ; sparse arrays - lower and upper triangular matrices and Tri-diagonal matrices; addition and subtraction of two sparse arrays. (Multidimensional, and, sparse arrays, to be given elementary treatment.)

Stacks and Queues: Introduction and primitive operations on stack; Stack application: Polish Notations; Evaluation of postfix expression; Conversion from infix to postfix; Introduction and primitive operations on queues; D-queues and priority queues.[T1,T2,T3]

[No. of Hrs: 11]

UNIT-II

Lists: Introduction to linked lists; Sequential and linked lists, operations such as traversal, insertion, deletion, searching, Two way lists and Use of headers

Trees: Introduction and terminology; Traversal of binary trees; Recursive algorithms for tree operations such as traversal, insertion and deletion; [T1, T2, T3]

[No. of Hrs: 11]

UNIT-III

Introduction to and creation of AVL trees and m-way search trees - (elementary treatment to be given); Multilevel indexing and B-Trees: Introduction; Indexing with binary search trees; Multilevel indexing, a better approach to tree indexes; Example for creating a B-tree. [T1, T2, T3]

[No. of Hrs: 11]

UNIT-IV

Sorting Techniques: Insertion sort, selection sort and merge sort.

Searching Techniques: linear search, binary search and hashing. (Complexities NOT to be discussed for sorting and searching) [T1, T2, T3]

[No. of Hrs: 11]

UNIT-I

Introduction to Data Structures : Basic Terminology

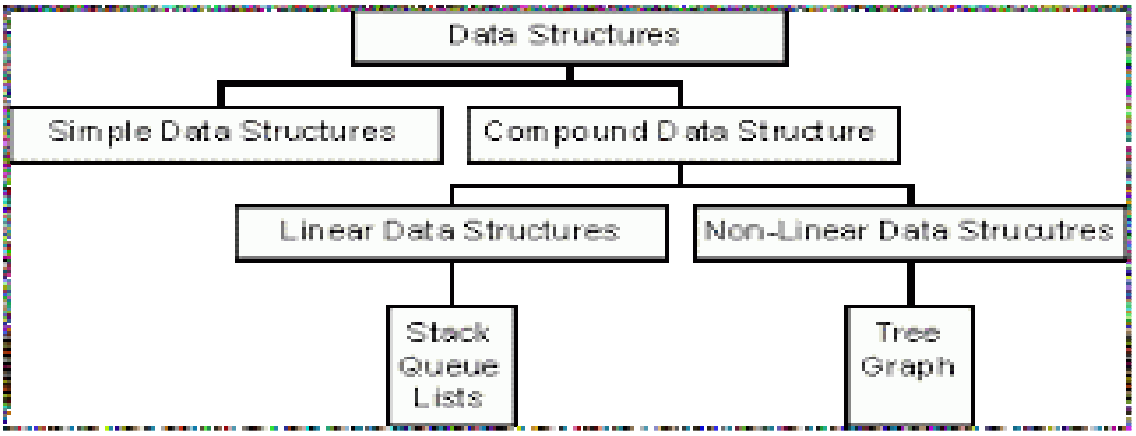
A container for the data is called data structure. A data structure is a means of organizing data in primary memory in a form that is convenient to process by a program. In the definition of data structure, structure means a set of rules that holds the data together. In other words, if we take a group of data and fit them into a structure such that we can define its relating rules, we have made a data structure. Data is stored either in main memory or in secondary memory. In order to represent data we need some models. The different models (logical or mathematical) to represent/organize/store data in the main memory are together referred to as data structures.
The data structure is collection of data organized in a specific manner in computer's main memory.

Elementary Data Organizations:

Data Types : The type of a data object in C determines the range and kind of values an object can represent, the size of machine storage reserved for an object, and the operations allowed on an object. Functions also have types, and the function's return type and parameter types can be specified in the function's declaration
Data Sizes : An object of a given data type is stored in a section of memory having a discreet size. Objects of different data types require different amounts of memory. Following table shows the size and range of the basic data types.

Data types	Size	Range
short int , or signed short int	16 bits	-32768 to 32767
unsigned short int	16 bits	0 to 65535
int or signed int	32 bits	-2147483648 to 2147483647
unsigned int	32 bits	0 to 4294967295
long int , or signed long int (OpenVMS)	32 bits	- 2147483648 to 2147483647
long int , or signed long int (Digital UNIX)	64 bits	-9223372036854775808 to 9223372036854775807
unsigned long int (OpenVMS)	32 bits	0 to 4294967295
unsigned long int (Digital UNIX)	64 bits	0 to 18446744073709551615
signed __int64 (Alpha)	64 bits	-9223372036854775808 to 9223372036854775807
unsigned __int64 (Alpha)	64 bits	0 to 18446744073709551615

Classification of data structures and its operations:



Data structures can be classified as

1. Simple data structure
2. Compound data structure
   2.1 Linear data structure
   2.2 Non linear data structure

Simple Data Structure: Simple data structure can be constructed with the help of primitive data structure. A primitive data structure used to represent the standard data types of any one of the computer languages. Variables, arrays, pointers, structures, unions, etc. are examples of primitive data structures.

Compound Data structure: Compound data structure can be constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user. It can be classified as

1) Linear data structure
2) Non-linear data structure

1) Linear data structure: Linear data structures can be constructed as a continuous arrangement of data elements in the memory. It can be constructed by using array data type. In the linear Data Structures the relationship of adjacency is maintained between the Data elements.

Operations applied on linear data structure:

The following list of operations applied on linear data structures

1. Add an element
2. Delete an element
3. Traverse
4. Sort the list of elements
5. Search for a data element

by applying one or more functionalities to create different types of data structures For example Stack, Queue,

Tables, List, and Linked Lists.

Non-linear data structure: Non-linear data structure can be constructed as a collection of randomly distributed set of data item joined together by using a special pointer (tag). In non-linear Data structure the relationship of adjacency is not maintained between the Data items.

Operations applied on non-linear data structures :
The following list of operations applied on non-linear data structures.
1. Add elements
2. Delete elements
3. Display the elements
4. Sort the list of elements
5. Search for a data element

By applying one or more functionalities and different ways of joining randomly distributed data items to create different types of data structures. For example Tree, Decision tree, Graph and Forest.

Arrays: Representation of single and multidimensional arrays (up to three dimensions) ;
Array is a collection of same data types occupying contiguous memory location.

Characteristics of Arrays in C
1) An array holds elements that have the same data type.
2) Array elements are stored in subsequent memory locations.
3) Two-dimensional array elements are stored row by row in subsequent memory locations.
4) Array name represents the address of the starting element.
5) Array size should be mentioned in the declaration. Array size must be a constant expression and not a variable.

Declaration of single dimensional Array
Arrays must be declared before they can be used in the program. Standard array declaration is as
Typevariable_name[lengthofarray];
Here type specifies the variable type of the element which is going to be stored in the array. In C programming language we can declare the array of any basic standard type which C language supports. For example

```
double height[10];
float width[20];
int min[9];
char name[20];
```

Declaration of multi dimensional Array:
```
type variable_name[size1][size2];

type variable_name[size1][size2][size3];
```

```
#include <stdio.h>
void oneWay(void);
```

```
void anotherWay(void);
int main(void) {
   printf("\noneWay:\n");
   oneWay();
   printf("\nantherWay:\n");
   anotherWay();
}
/*Array initialized with aggregate */
void oneWay(void) {
   int vect[10] = {1,2,3,4,5,6,7,8,9,0};
   int i;
   for (i=0; i<10; i++){
      printf("i = %2d vect[i] = %2d\n", i, vect[i]);
   }
}
/*Array initialized with loop */
void anotherWay(void) {
   int vect[10];
   int i;
   for (i=0; i<10; i++)
       vect[i] = i+1;
   for (i=0; i<10; i++)
       printf("i = %2d vect[i] = %2d\n", i, vect[i]);
}
```

The output of this program is

```
   oneWay:
   i = 0 vect[i] = 1
   i = 1 vect[i] = 2
   i = 2 vect[i] = 3
   i = 3 vect[i] = 4
   i = 4 vect[i] = 5
   i = 5 vect[i] = 6
   i = 6 vect[i] = 7
   i = 7 vect[i] = 8
   i = 8 vect[i] = 9
   i = 9 vect[i] = 0

   antherWay:
   i = 0 vect[i] = 1
   i = 1 vect[i] = 2
   i = 2 vect[i] = 3
   i = 3 vect[i] = 4
   i = 4 vect[i] = 5
   i = 5 vect[i] = 6
   i = 6 vect[i] = 7
   i = 7 vect[i] = 8
   i = 8 vect[i] = 9
   i = 9 vect[i] = 10
```

Sparse arrays:

An array in which most of the elements are zero is called as a sparse array.

Example of sparse matrix

[1 2 0 0 0 0 0]
[0 3 4 0 0 0 0]
[0 0 5 6 7 0 0]
[0 0 0 0 0 8 0]
[0 0 0 0 0 0 9]

lower and upper triangular matrices and Tri-diagonal matrices;

A Lower triangular matrix is a matrix that has nonzero elements only in lower part of the main diagonal.
For example, the following matrix is Lower triangular matrix.

0 0 0 0
1 0 0 0
1 1 0 0
1 1 1 0

A upper triangular matrix is a matrix that has nonzero elements only in upper part of the main diagonal.
For example, the following matrix is upper triangular matrix

0 1 1 1
0 0 1 1
0 0 0 1
0 0 0 0

A tridiagonal matrix is a matrix that has nonzero elements only in the main diagonal, the first diagonal below this, and the first diagonal above the main diagonal.
For example, the following matrix is tridiagonal:

$$\begin{pmatrix} 1 & 4 & 0 & 0 \\ 3 & 4 & 1 & 0 \\ 0 & 2 & 3 & 1 \\ 0 & 0 & 1 & 3 \end{pmatrix}.$$

Algorithm :Addition of two sparse arrays.

1. Initialize
   l=1
   T=0
2. Scan each row
   Repeat thru step 9 while l<=M
3. Obtain row indices and starting positions of next rows
   J=AROW[l]
   K=BROW[l]

```
      CROW[l]=T+1
       AMAX=BMAX=0
       If l<M
       then Repeat for P=l+1, l+2, .....M while AMAX=0
              If AROW[P]/=0
              then AMAX=AROW[P]
            Repeat for P=l+1, l+2,.....M while BMAX=0
              If BROW[P]/=0
              then BMAX=BROW[P]
       If AMAX=0
       then AMAX=R+1
       If BMAX=0
       then BMAX=S+1
4. Scan columns of this row
       Repeat thru step 7 while J/=0 and K/=0
5. Elements in same column?
       If ACOL[J]=BCOL[K]
       then SUM=A[J]+B[K]
       COLUMN=ACOL[J]
       J=J+1
       K=K+1
  else If ACOL[J]<BCOL[K]
       then SUM=A[J]
             COLUMN=ACOL[J]
             J=J+1
       else SUM=B[K]
            COLUMN=BCOL[K]
            K=K+1
6. Add new elements to sum of matrices
       If SUM/=0
       then T=T+1
            C[T]=SUM
            CCOL[T]=COLUMN
7. End of either row?
       If J=AMAX
       then J=0
       If K=BMAX
       then K=0
8. Add remaining elements of a row
       If J=0 and K/=0
       then repeat while K<BMAX
            T=T+1
            C[T]=B[K]
            CCOL[T]=BCOL[K]
            K=K+1
       else if K=0 and J/=0
           then repeat while J<AMAX
                T=T+1
```

C[T]=A[J]
CCOL[T]=ACOL[J]
J=J+1

9. Adjust index to matrix C and increment row index
   If T<CROW[l]
   then CROW[l]=0
   l=l+1
10. Finished
   Exit

Algorithm : Subtraction of two sparse arrays.

1. Initialize
   l=1
   T=0
2. Scan each row
   Repeat thru step 9 while l<=M
3. Obtain row indices and starting positions of next rows
   J=AROW[l]
   K=BROW[l]
   CROW[l]=T+1
   AMAX=BMAX=0
   If l<M
   then Repeat for P=l+1, l+2, ...M while AMAX=0
      If AROW[P]/=0
      then AMAX=AROW[P]
   Repeat for P=l+1, l+2,.....M while BMAX=0
      If BROW[P]/=0
      then BMAX=BROW[P]
   If AMAX=0
   then AMAX=R+1
   If BMAX=0
   then BMAX=S+1
4. Scan columns of this row
   Repeat thru step 7 while J/=0 and K/=0
5. Elements in same column?
   If ACOL[J]=BCOL[K]
   then SUB=A[J]-B[K]
   COLUMN=ACOL[J]
   J=J+1
   K=K+1
 else If ACOL[J]<BCOL[K]
   then SUM=A[J]
      COLUMN=ACOL[J]
      J=J+1
   else SUM=B[K]
      COLUMN=BCOL[K]
      K=K+1

6. Add new elements to sum of matrices
   If SUM/=0
   then T=T+1
      C[T]=SUM
      CCOL[T]=COLUMN
7. End of either row?
   If J=AMAX
   then J=0
   If K=BMAX
   then K=0
8. Add remaining elements of a row
   If J=0 and K/=0
   then repeat while K<BMAX
      T=T+1
      C[T]=B[K]
      CCOL[T]=BCOL[K]
      K=K+1
   else if K=0 and J/=0
      then repeat while J<AMAX
         T=T+1
         C[T]=A[J]
         CCOL[T]=ACOL[J]
         J=J+1
9. Adjust index to matrix C and increment row index
   If T<CROW[l]
   then CROW[l]=0
   l=l+1
10. Finished
   Exit

Difference between array and linked list

1. Array is collection of homogeneous elements. List is collection of heterogeneous elements.
2. Array, memory allocated is static and continuous. For List, memory allocated is dynamic and random.
3. Array : User need not have to keep in track of next memory allocation. List : User has to keep in Track of next location where memory is allocated.

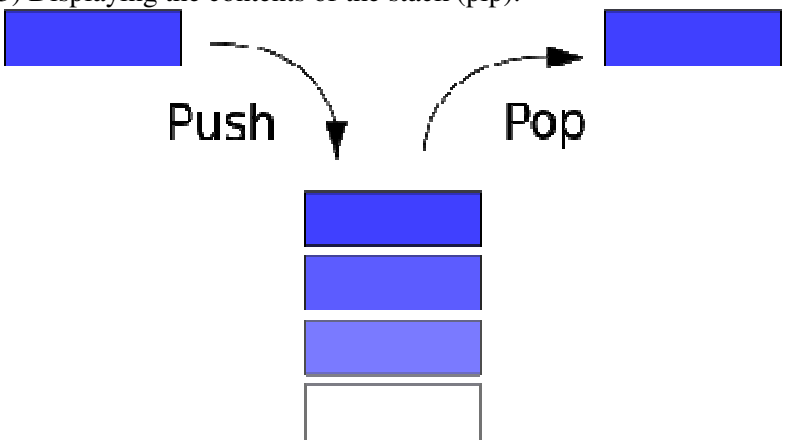
For Example: Array uses direct access of stored members; list uses sequential access for members. //With Array you have direct access to memory position 5 Object x = a[5]; // x takes directly a reference to 5th element of array //With the list you have to cross all previous nodes in order to get the 5th node: list mylist; list::iterator it; for(it = list.begin() ; it != list.end() ; it++) { if(i==5) { x = *it; break; } i++; }

Stacks : Introduction and primitive operations on stack;

A stack is a basic data structure, where insertion and deletion of items takes place at one end called top of the stack. The basic concept can be illustrated by thinking of your data as a stack of plates or books where you can only take the top item off the stack in order to remove things from it. A stack is also called a LIFO (Last In First Out) to demonstrate the way it accesses data.

There are basically three operations that can be performed on stacks.

1) Inserting an item into a stack (push).
2) Deleting an item from the stack (pop).
3) Displaying the contents of the stack (pip).



Stack application: Polish Notations

Polish notation, also known as Polish prefix notation or simply prefix notation, is a form of notation for logic, arithmetic, and algebra. Its distinguishing feature is that it places operators to the left of their operands.
Example: +ab

Evaluation of postfix expression

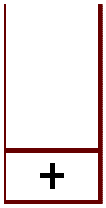
Example: An equivalent in-fix is as follows: ((15 / (7 − (1 + 1))) * 3) − (2 + (1 + 1)) = 5

− * / 15 − 7 + 1 1 3 + 2 + 1 1 =
− * / 15 − 7 2 3 + 2 + 1 1 =
− * / 15 5 3 + 2 + 1 1 =
− * 3 3 + 2 + 1 1 =
− 9 + 2 + 1 1 =
− 9 + 2 2 =
− 9 4 =
5

Conversion from infix to postfix;

Example :
Infix String : a+b*c-d

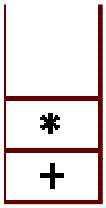
Initially the Stack is empty and our Postfix string has no characters. Now, the first character scanned is 'a'. 'a' is added to the Postfix string. The next character scanned is '+'. It being an operator, it is pushed to the stack.



Stack

Postfix String

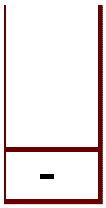
Next character scanned is 'b' which will be placed in the Postfix string. Next character is '*' which is an operator. Now, the top element of the stack is '+' which has lower precedence than '*', so '*' will be pushed to the stack.



Stack

Postfix String

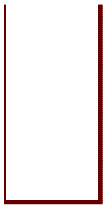
The next character is 'c' which is placed in the Postfix string. Next character scanned is '-'. The topmost character in the stack is '*' which has a higher precedence than '-'. Thus '*' will be popped out from the stack and added to the Postfix string. Even now the stack is not empty. Now the topmost element of the stack is '+' which has equal priority to '-'. So pop the '+' from the stack and add it to the Postfix string. The '-' will be pushed to the stack.



Stack

Postfix String

Next character is 'd' which is added to Postfix string. Now all characters have been scanned so we must pop the remaining elements from the stack and add it to the Postfix string. At this stage we have only a '-' in the stack. It is popped out and added to the Postfix string. So, after all characters are scanned, this is how the stack and Postfix string will be :


Stack

Postfix String

End result :

- Infix String : a+b*c-d
- Postfix String : abc*+d-

Introduction and primitive operations on queues; A queue is a list in which insertion can be perform at rare end of queue and deletion can be perform from front end of queue. It's basically works on act of FIFO (First in, First out).

Basic Operations on Queue
- Insert
- Delete
- Top (Returns topmost element of queue)
- IsEmpty (Which returns TRUE if queue is empty)

Properties of Queue
- Insertion is at last and Deletion is of first element from the list.
- Top of queue points to first in element.
- Queue can be static or dynamic in size.
- The indexing of particular element of queue depends on the basic list which we have use to implement it.

Priority Queue :
A Priority queue is a collection of zero or more elements. Each element has a priority or a value. The operations performed on a priority queue are :
1) Find an element
2) Insert a new element
3) Delete an element In "Min Priority Queue" the find operation finds the element with minimum priority, while the delete operation deletes it. In "Max Priority Queue" the find operation finds the element with maximum priority, while the delete operation deletes it. Unlike the general queues which are FIFO structures, the order of deletion from a priority queue is determined by the element priority. Elements are deleted either in increasing or decreasing order of priority from a priority queue.
DeQueue: This differs from the queue abstract data type or First-In-First-Out List (FIFO), where elements can only be added to one end and removed from the other. This general data class has some possible sub-types:

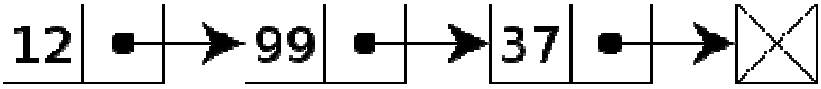
- An input-restricted deque is one where deletion can be made from both ends, but insertion can be made at one end only.
- An output-restricted deque is one where insertion can be made at both ends, but deletion can be made from one end only.

UNIT-II

Lists: Introduction to linked lists: A linked list is a most common data structure. That is not take contiguous memory location. A linked list can be divided into different types.

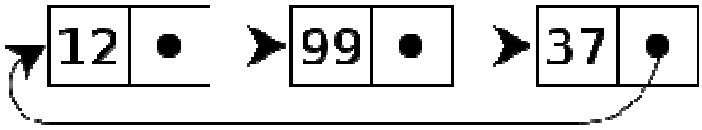
1. Singly linked list
2. Doubly linked list
3. Circular list

Singly linked list: Singly linked lists contain nodes which have a data field as well as a next field, which points to the next node in the linked list.



A singly linked list whose nodes contain two fields: an integer value and a link to the next node
Doubly linked list: In a doubly linked list, each node contains, besides the next-node link, a second link field pointing to the previous node in the sequence. The two links may be called forward(s) and backwards, or next and previous.

A doubly linked list whose nodes contain three fields: an integer value, the link forward to the next node, and the link backward to the previous node
Circular list: In the last node of a list, the link field often contains a null reference, a special value used to indicate the lack of further nodes. A less common convention is to make it point to the first node of the list; in that case the list is said to be circular or circularly linked; otherwise it is said to be open or linear.



A circular linked list
In the case of a circular doubly linked list, the only change that occurs is that the end, or "tail", of the said list is linked back to the front, or "head", of the list and vice versa.

Operations such as traversal, insertion, deletion, searching:

The following program shows how a simple, linear linked list can be constructed in C, using dynamic memory allocation and pointers.

```
#include<stdlib.h>
#include<stdio.h>

struct list_el {
  int val;
  struct list_el * next;
};
```

```
typedef struct list_el item;

void main() {
  item * curr, * head;
  int i;

  head = NULL;

  for(i=1;i<=10;i++) {
    curr = (item *)malloc(sizeof(item));
    curr->val = i;
    curr->next  = head;
    head = curr;
  }

  curr = head;

  while(curr) {
    printf("%d\n", curr->val);
    curr = curr->next ;
  }
}
```

Adding(Inserting) a node to a singly linked list has only two cases:
1. *head = ;* in which case the node we are adding is now both the *head* and *tail* of the list; or
2. we simply need to append our node onto the end of the list updating the *tail* reference appropriately.

1) algorithm Add(*value*)
2) Pre: *value* is the value to add to the list
3) Post: *value* has been placed at the tail of the list
4) *n Ã* node(*value*)
5) if *head = ;*
6) *head Ã n*
7) *tail Ã n*
8) else
9) *tail*.Next *Ã n*
10) *tail Ã n*
11) end if
12) end Add

Searching:

1) algorithm Contains(*head*, *value*)
2) Pre: *head* is the head node in the list
3) *value* is the value to search for
4) Post: the item is either in the linked list, true; otherwise false
5) *n Ã head*
6) while *n 6= ;* and *n*.Value *6= value*

7) *n Ã n*.Next
8) end while
9) if *n = ;*
10) return false
11) end if
12) return true
13) end

Deletion:
1) algorithm Remove(*head*, *value*)
2) Pre: *head* is the head node in the list
3) *value* is the value to remove from the list
4) Post: *value* is removed from the list, true; otherwise false
5) if *head = ;*
6) // case 1
7) return false
8) end if
9) *n Ã head*
10) if *n*.Value = *value*
11) if *head = tail*
12) // case 2
13) *head Ã ;*
14) *tail Ã ;*
15) else
16) // case 3
17) *head Ã head*.Next
18) end if
19) return true
20) end if
21) while *n*.Next *6= ;* and *n*.Next.Value *6= value*
22) *n Ã n*.Next
23) end while
24) if *n*.Next *6= ;*
25) if *n*.Next = *tail*
26) // case 4
27) *tail Ã n*
28) end if
29) // this is only case 5 if the conditional on line 25 was *false*
30) *n*.Next *Ã n*.Next.Next
31) return true
32) end if
33) // case 6
34) return false
35) end Remove

Traversal:
1) algorithm Traverse(*head*)
2) Pre: *head* is the head node in the list

3) Post: the items in the list have been traversed
4) *n Ã head*
5) while *n 6*= 0
6) yield *n*.Value
7) *n Ã n*.Next
8) end while
9) end Traverse

Reverse traversal:
1) algorithm ReverseTraversal(*head*, *tail*)
2) Pre: *head* and *tail* belong to the same list
3) Post: the items in the list have been traversed in reverse order
4) if *tail 6*= *;*
5) *curr Ã tail*
6) while *curr 6*= *head*
7) *prev Ã head*
8) while *prev*.Next *6*= *curr*
9) *prev Ã prev*.Next
10) end while
11) yield *curr*.Value
12) *curr Ã prev*
13) end while
14) yield *curr*.Value
15) end if
16) end ReverseTraversal

Two way linked list:
Insertion:
1) algorithm Add(*value*)
2) Pre: *value* is the value to add to the list
3) Post: *value* has been placed at the tail of the list
4) *n Ã* node(*value*)
5) if *head* = *;*
6) *head Ã n*
7) *tail Ã n*
8) else
9) *n*.Previous *Ã tail*
10) *tail*.Next *Ã n*
11) *tail Ã n*
12) end if
13) end Add

Deletion:
1) algorithm Remove(*head*, *value*)
2) Pre: *head* is the head node in the list
3) *value* is the value to remove from the list
4) Post: *value* is removed from the list, true; otherwise false

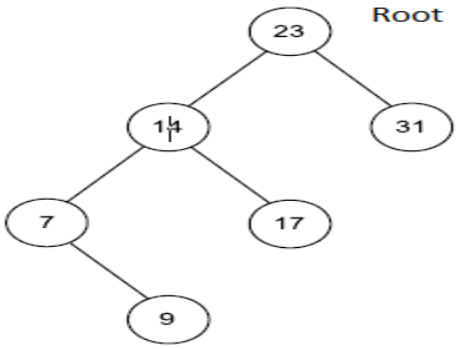
5) if *head* = *;*
6) return false
7) end if
8) if *value* = *head*.Value
9) if *head* = *tail*
10) *head Ã ;*
11) *tail Ã ;*
12) else
13) *head Ã head*.Next
14) *head*.Previous *Ã ;*
15) end if
16) return true
17) end if
18) *n Ã head*.Next
19) while *n 6*= *;* and *value 6*= *n*.Value
20) *n Ã n*.Next
21) end while
22) if *n* = *tail*
23) *tail Ã tail*.Previous
24) *tail*.Next *Ã ;*
25) return true
26) else if *n 6*= *;*
27) *n*.Previous.Next *Ã n*.Next
28) *n*.Next.Previous *Ã n*.Previous
29) return true
30) end if
31) return false
32) end Remove

Reverse Traversal:
1) algorithm ReverseTraversal(*tail*)
2) Pre: *tail* is the tail node of the list to traverse
3) Post: the list has been traversed in reverse order
4) *n Ã tail*
5) while *n 6*= *;*
6) yield *n*.Value
7) *n Ã n*.Previous
8) end while
9) end ReverseTraversal

Use of headers: Header node is needed to keep track of the node of the linked list this is a start node for traversing ,displaying ,insertion, deletion.

Trees: Introduction and terminology: A tree is a non-empty collection of vertices & edges that satisfies certain requirements. A vertex is a simple object (node) that can have a name and carry other associated information. An edge is a connection between two vertices. A Tree is a finite set of a zero or more vertices such that there is one specially designated vertex called Root and the remaining vertices are partitioned into a

collection of sub-trees, each of which is also a tree. A node may not have children, such node is known as Leaf (terminal node). The line from parent to a child is called a branch or an edge. Children to same parent are called siblings.



Traversal of binary trees; Recursive algorithms for tree operations such as traversal, insertion and deletion;

Indexing with Binary search Tree: In computer science, a binary tree is a tree data structure in which each node has at most two children. Typically the child nodes are called left and right. A binary tree consists of a node (called the root node) and left and right sub-trees.

Both the sub-trees are themselves binary trees. The nodes at the lowest levels of the tree (the ones with no sub-trees) are called leaves. In an ordered binary tree,
(a) The keys of all the nodes in the left sub-tree are less than that of the root.
(b) The keys of all the nodes in the right sub-tree are greater than that of the root.
(c) The left and right sub-trees are themselves ordered binary trees.

Tree traversal:
Preorder:
1) algorithm Preorder(*root*)
2) Pre: *root* is the root node of the BST
3) Post: the nodes in the BST have been visited in preorder
4) if *root* 6= *;*
5) yield *root*.Value
6) Preorder(*root*.Left)
7) Preorder(*root*.Right)
8) end if
9) end Preorder

Postorder:
1) algorithm Postorder(*root*)
2) Pre: *root* is the root node of the BST
3) Post: the nodes in the BST have been visited in postorder
4) if *root* 6= *;*
5) Postorder(*root*.Left)
6) Postorder(*root*.Right)
7) yield *root*.Value

8) end if
9) end Postorder

Inorder:
1) algorithm Inorder(*root*)
2) Pre: *root* is the root node of the BST
3) Post: the nodes in the BST have been visited in inorder
4) if *root 6*= *;*
5) Inorder(*root*.Left)
6) yield *root*.Value
7) Inorder(*root*.Right)
8) end if
9) end Inorder

Insertion:
1) algorithm Insert(*value*)
2) Pre: *value* has passed custom type checks for type *T*
3) Post: *value* has been placed in the correct location in the tree
4) if *root* = *;*
5) *root Ã* node(*value*)
6) else
7) InsertNode(*root*, *value*)
8) end if
9) end Insert
1) algorithm InsertNode(*current*, *value*)
2) Pre: *current* is the node to start from
3) Post: *value* has been placed in the correct location in the tree
4) if *value* < *current*.Value
5) if *current*.Left = *;*
6) *current*.Left *Ã* node(*value*)
7) else
8) InsertNode(*current*.Left, *value*)
9) end if
10) else
11) if *current*.Right = *;*
12) *current*.Right *Ã* node(*value*)
13) else
14) InsertNode(*current*.Right, *value*)
15) end if
16) end if
17) end InsertNode

Searching:
1) algorithm Contains(*root*, *value*)
2) Pre: *root* is the root node of the tree, *value* is what we would like to locate
3) Post: *value* is either located or not
4) if *root* = *;*
5) return false

6) end if
7) if *root*.Value = *value*
8) return true
9) else if *value* < *root*.Value
10) return Contains(*root*.Left, *value*)
11) else
12) return Contains(*root*.Right, *value*)
13) end if
14) end Contains

Deletion:
1) algorithm Remove(*value*)
2) Pre: *value* is the value of the node to remove, *root* is the root node of the BST
3) Count is the number of items in the BST
3) Post: node with *value* is removed if found in which case yields true, otherwise false
4) *nodeToRemove Ã* FindNode(*value*)
5) if *nodeToRemove* = *;*
6) return false // value not in BST
7) end if
8) *parent Ã* FindParent(*value*)
9) if Count = 1
10) *root Ã ;* // we are removing the only node in the BST
11) else if *nodeToRemove*.Left = *;* and *nodeToRemove*.Right = *null*
12) // case #1
13) if *nodeToRemove*.Value < *parent*.Value
14) *parent*.Left *Ã ;*
15) else
16) *parent*.Right *Ã ;*
17) end if
18) else if *nodeToRemove*.Left = *;* and *nodeToRemove*.Right *6*= *;*
19) // case # 2
20) if *nodeToRemove*.Value < *parent*.Value
21) *parent*.Left *Ã nodeToRemove*.Right
22) else
23) *parent*.Right *Ã nodeToRemove*.Right
24) end if
25) else if *nodeToRemove*.Left *6*= *;* and *nodeToRemove*.Right = *;*
26) // case #3
27) if *nodeToRemove*.Value < *parent*.Value
28) *parent*.Left *Ã nodeToRemove*.Left
29) else
30) *parent*.Right *Ã nodeToRemove*.Left
31) end if
32) else
33) // case #4
34) *largestV alue Ã nodeToRemove*.Left
35) while *largestV alue*.Right *6*= *;*
36) // ¯nd the largest value in the left subtree of *nodeToRemove*

37) *largestV alue Ã largestV alue*.Right
38) end while
39) // set the parents' Right pointer of *largestV alue* to *;*
40) FindParent(*largestV alue*.Value).Right *Ã ;*
41) *nodeToRemove*.Value *Ã largestV alue*.Value
42) end if
43) Count *Ã* Count ¡1
44) return true
45) end Remove

UNIT-III

Introduction to and creation of AVL trees:

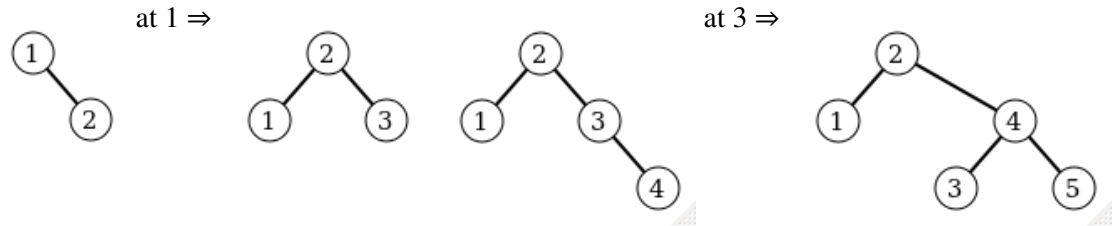
An AVL tree is a self-balancing binary search tree, and the first such data structure to be invented. In an AVL tree the heights of the two child subtrees of any node differ by at most one, therefore it is also called height-balanced. Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases. Additions and deletions may require the tree to be rebalanced by one or more tree rotations.

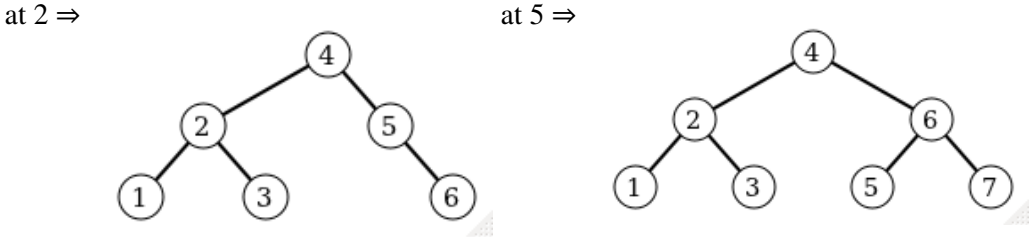
The AVL tree is named after its two inventors, G.M. Adelson-Velsky and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information." The balance factor of a node is the height of its right subtree minus the height of its left subtree. A node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees.

An important example of AVL trees is

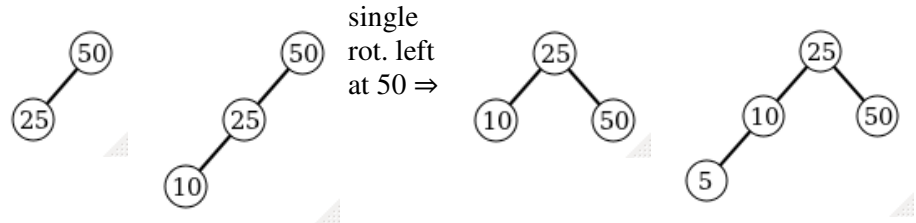
1, 2, 3, 4, 5, 6, 7

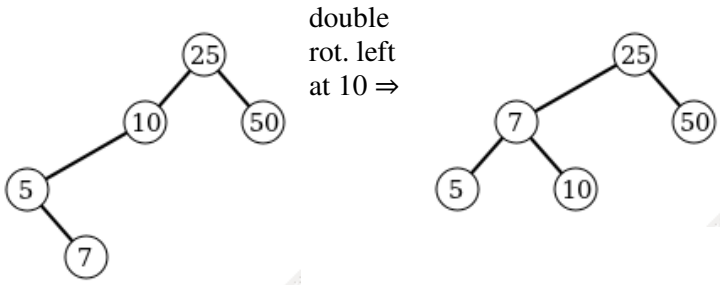
All insertions are right-right and so rotations are all single rotate from the right. All but two insertions require rebalancing:

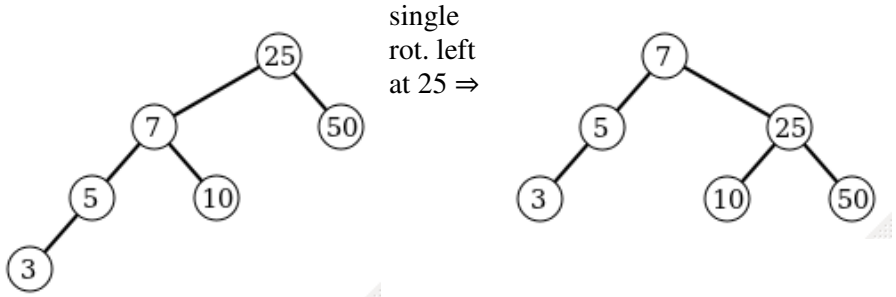




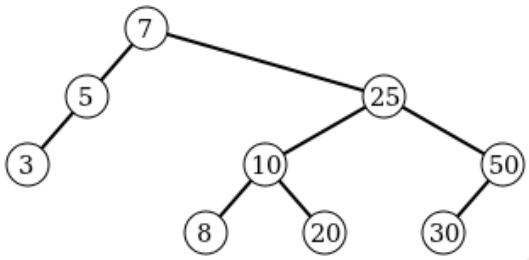
Example2: The insertion sequence is: 50, 25, 10, 5, 7, 3, 30, 20, 8, 15

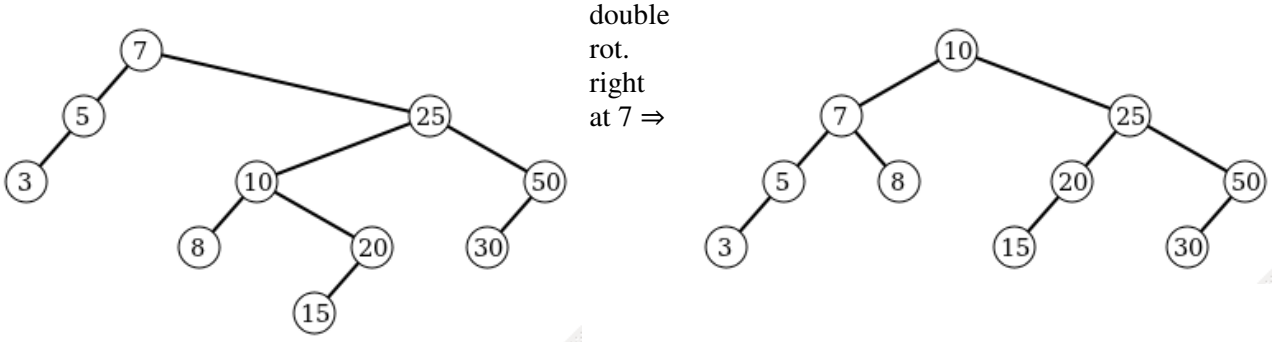






add(30), add(20), add(8) need no rebalancing





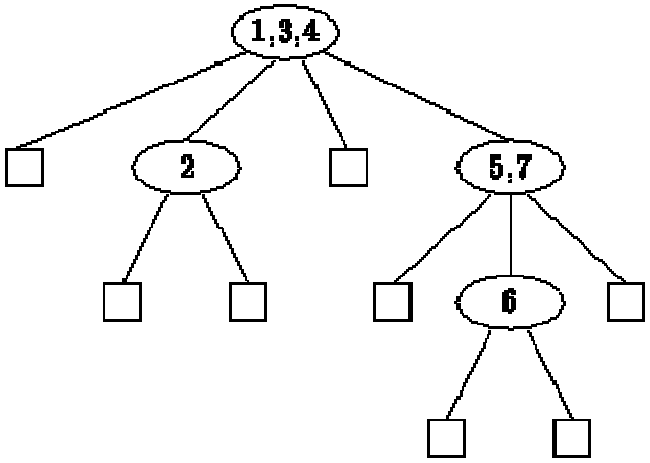
M-way search trees, Multilevel indexing and B-Trees: Introduction; Indexing with binary search trees; Multilevel indexing, a better approach to tree indexes; Example for creating a B-tree.

M-way search tree or B tree is a tree with n order. If an M-way tree of order then the maximum number of keys will be 4 and minimum number of keys will be 2.

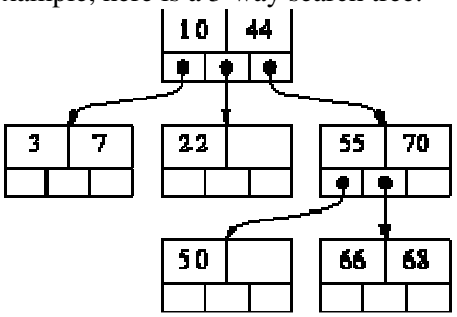
Definition (*M*-way Search Tree) An *M-way search tree* T is a finite set of keys. Either the set is empty, $T = \emptyset$; or the set consists of n M-way subtrees $T_0, T_1, \ldots, T_{n-1}$, and n-1 keys, $k_1, k_2, \ldots, k_{n-1}$, $T = \{T_0, k_1, T_1, k_2, T_2, \ldots, k_{n-1}, T_{n-1}\}$, where $2 \leq n \leq M$, such that the keys and nodes satisfy the following *data ordering properties* :

1. The keys in each node are distinct and ordered, i.e., $k_i < k_{i+1}$ for $1 \leq i \leq n-1$.
2. All the keys contained in subtree T_{i-1} are less than k_i, i.e., $\forall k \in T_{i-1} : k < k_i$ for $1 \leq i \leq n-1$. The tree T_{i-1} is called the *left subtree* with respect to the key k_i.
3. All the keys contained in subtree T_i are greater than k_i, i.e., $\forall k \in T_i : k > k_i$ for $1 \leq i \leq n-1$. The tree T_{i+1} is called the *right subtree* with respect to the key k_i.

Figure gives an example of an *M*-way search tree for *M*=4. In this case, each of the non-empty nodes of the tree has between one and three keys and at most four subtrees. All the keys in the tree satisfy the data ordering properties. Specifically, the keys in each node are ordered and for each key in the tree, all the keys in the left subtree with respect to the given key are less than the given key, and all the keys in the right subtree with respect to the given key are larger than the given key. Finally, it is important to note that the topology of the tree is not determined by the particular set of keys it contains.



For example, here is a 3-way search tree:

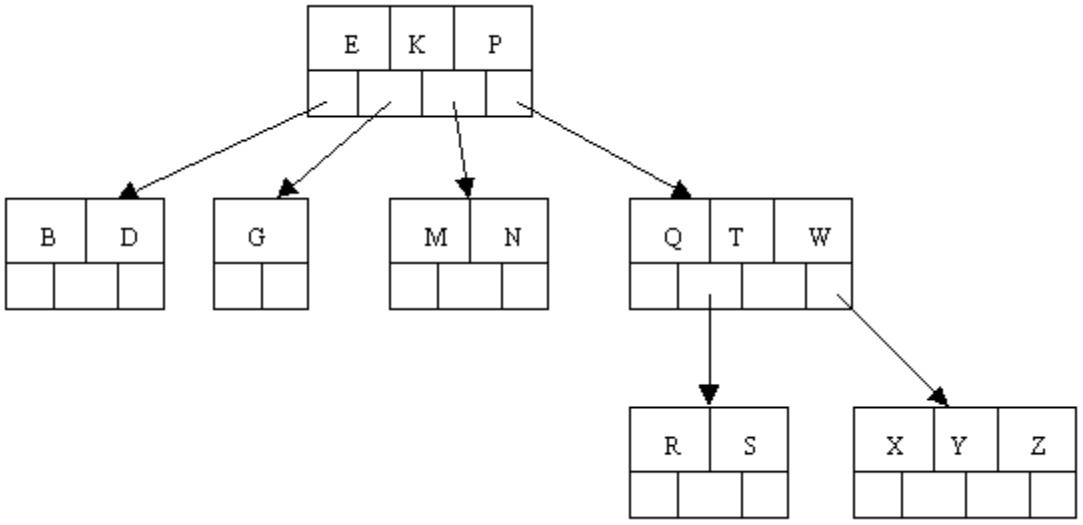


B-Trees Introduction:

A B-tree is a specialized multiway tree designed especially for use on disk. In a B-tree each node may contain a large number of keys. The number of subtrees of each node, then, may also be large. A B-tree is designed to branch out in this large number of directions and to contain a lot of keys in each node so that the height of the tree is relatively small. This means that only a small number of nodes must be read from disk to retrieve an item. The goal is to get fast access to the data, and with disk drives this means reading a very small number of records. Note that a large node size (with lots of keys in the node) also fits with the fact that with a disk drive one can usually read a fair amount of data at once.

Definitions

A *multiway tree* *of order m* is an ordered tree where each node has at most m children. For each node, if k is the actual number of children in the node, then k - 1 is the number of keys in the node. If the keys and subtrees are arranged in the fashion of a search tree, then this is called a *multiway search tree of order m*. For example, the following is a multiway search tree of order 4. Note that the first row in each node shows the keys, while the second row shows the pointers to the child nodes. Of course, in any useful application there would be a record of data associated with each key, so that the first row in each node might be an array of records where each record contains a key and its associated data. Another approach would be to have the first row of each node contain an array of records where each record contains a key and a record number for the associated data record, which is found in another file. This last method is often used when the data records are large. The example software will use the first method.



What does it mean to say that the keys and subtrees are "arranged in the fashion of a search tree"? Suppose that we define our nodes as follows:

```
typedef struct
  {
  int Count;        // number of keys stored in the current node
  ItemType Key[3];  // array to hold the 3 keys
  long Branch[4];   // array of fake pointers (record numbers)
  } NodeType;
```

Then a multiway search tree of order 4 has to fulfill the following conditions related to the ordering of the keys:

- The keys in each node are in ascending order.
- At every given node (call it Node) the following is true:

- The subtree starting at record Node. Branch[0] has only keys that are less than Node. Key[0].
- The subtree starting at record Node. Branch[1] has only keys that are greater than Node. Key[0] and at the same time less than Node.Key[1].
- The subtree starting at record Node. Branch[2] has only keys that are greater than Node. Key[1] and at the same time less than Node.Key[2].
- The subtree starting at record Node. Branch[3] has only keys that are greater than Node. Key[2].

- Note that if less than the full number of keys is in the Node; these 4 conditions are truncated so that they speak of the appropriate number of keys and branches.

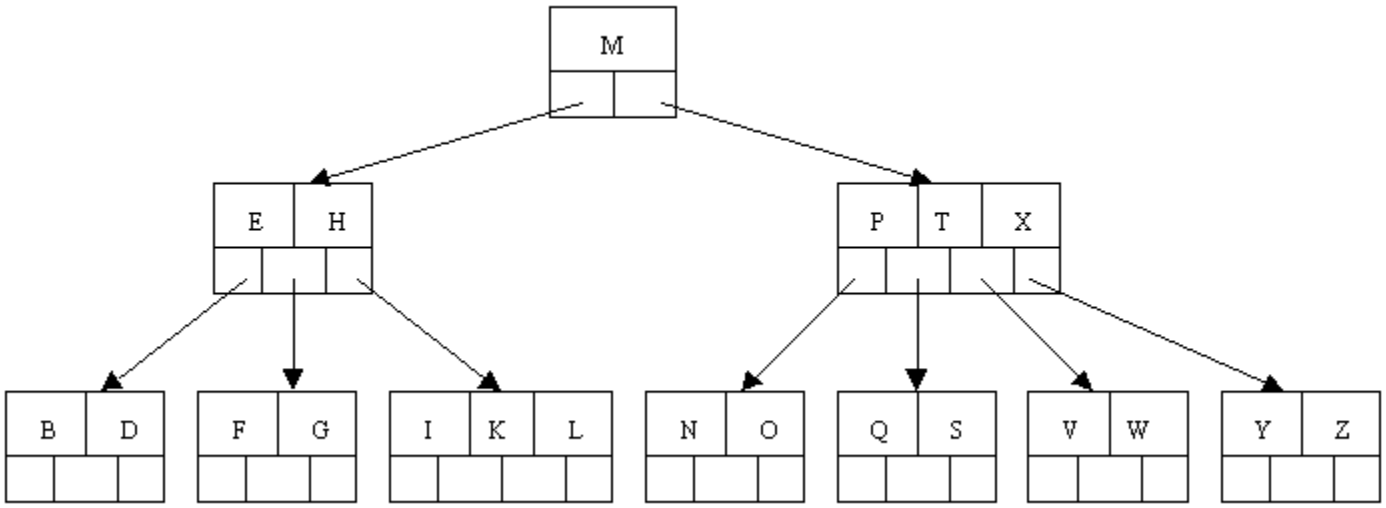
This generalizes in the obvious way to multiway search trees with other orders.
A B-tree of order m is a multiway search tree of order m such that:

- All leaves are on the bottom level.
- All internal nodes (except the root node) have at least ceil(m / 2) (nonempty) children.
- The root node can have as few as 2 children if it is an internal node, and can obviously have no children if the root node is a leaf (that is, the whole tree consists only of the root node).
- Each leaf node must contain at least ceil(m / 2) - 1 keys.

Note that ceil(x) is the so-called ceiling function. It's value is the smallest integer that is greater than or equal to x. Thus ceil(3) = 3, ceil(3.35) = 4, ceil(1.98) = 2, ceil(5.01) = 6, ceil(7) = 7, etc.
A B-tree is a fairly well-balanced tree by virtue of the fact that all leaf nodes must be at the bottom. Condition (2) tries to keep the tree fairly bushy by insisting that each node have at least half the maximum number of children. This causes the tree to "fan out" so that the path from root to leaf is very short even in a tree that contains a lot of data.

Example B-Tree multilevel Indexing

The following is an example of a B-tree of order 5. This means that (other that the root node) all internal nodes have at least ceil(5 / 2) = ceil(2.5) = 3 children (and hence at least 2 keys). Of course, the maximum number of children that a node can have is 5 (so that 4 is the maximum number of keys). According to condition 4, each leaf node must contain at least 2 keys. In practice B-trees usually have orders a lot bigger than 5.



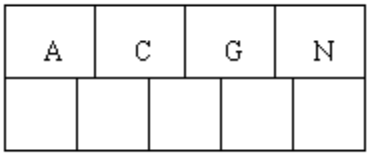
Operations on a B-Tree

Question: How would you search in the above tree to look up S? How about J? How would you do a sort-of "in-order" traversal, that is, a traversal that would produce the letters in ascending order? (One would only do such a traversal on rare occasion as it would require a large amount of disk activity and thus be very slow!)

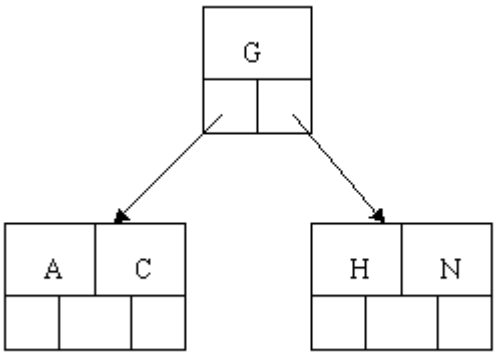
Inserting a New Item

When inserting an item, first do a search for it in the B-tree. If the item is not already in the B-tree, this unsuccessful search will end at a leaf. If there is room in this leaf, just insert the new item here. Note that this may require that some existing keys be moved one to the right to make room for the new item. If instead this leaf node is full so that there is no room to add the new item, then the node must be "split" with about half of the keys going into a new node to the right of this one. The median (middle) key is moved up into the parent node. (Of course, if that node has no room, then it may have to be split as well.) Note that when adding to an internal node, not only might we have to move some keys one position to the right, but the associated pointers have to be moved right as well. If the root node is ever split, the median key moves up into a new root node, thus causing the tree to increase in height by one.

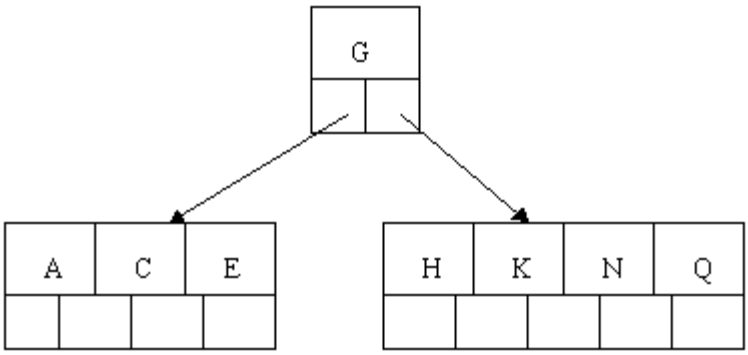
Let's work our way through an example similar to that given by Kruse. Insert the following letters into what is originally an empty B-tree of order 5: C N G A H E K Q M F W L T Z D P R X Y S Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys. The first 4 letters get inserted into the same node, resulting in this picture:



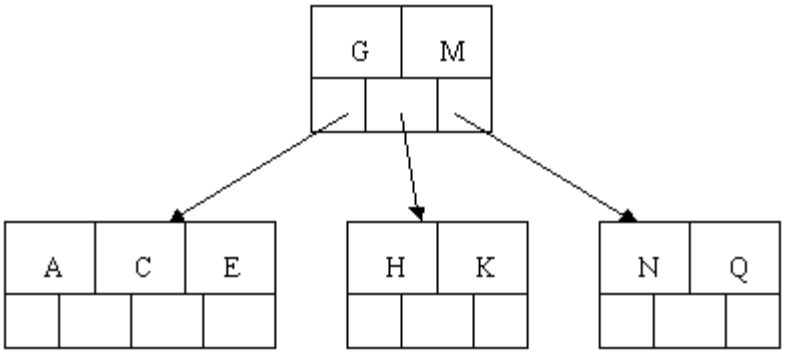
When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node. Note that in practice we just leave the A and C in the current node and place the H and N into a new node to the right of the old one.



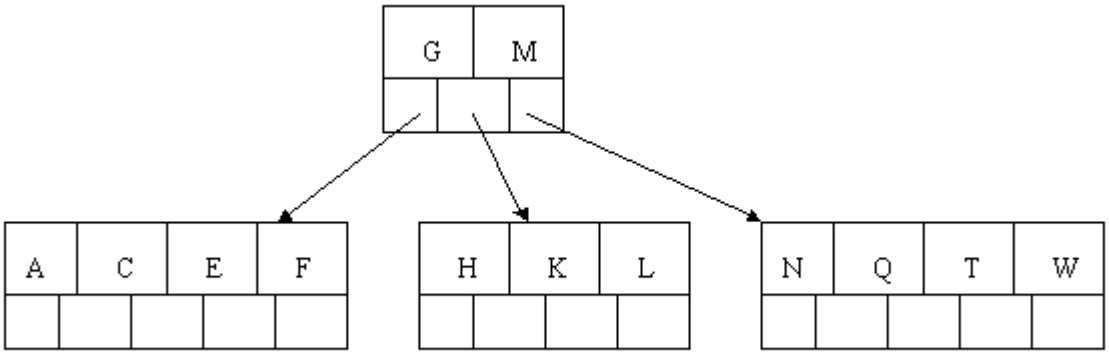
Inserting E, K, and Q proceeds without requiring any splits:



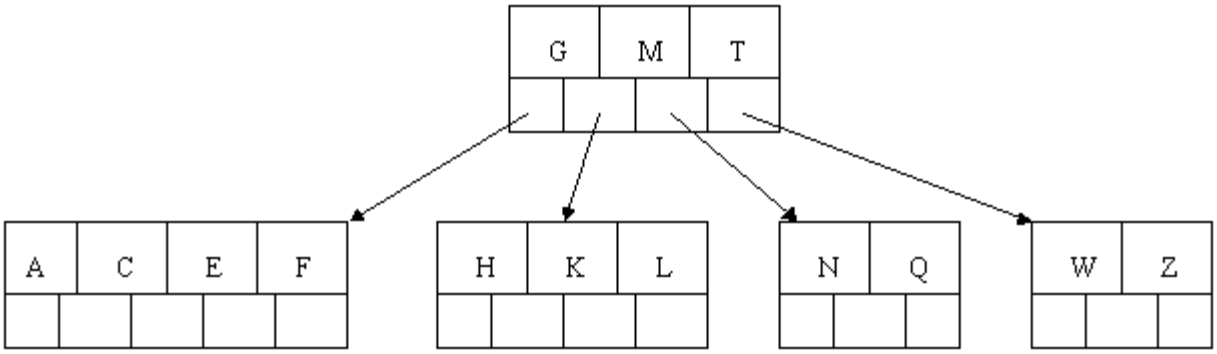
Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.



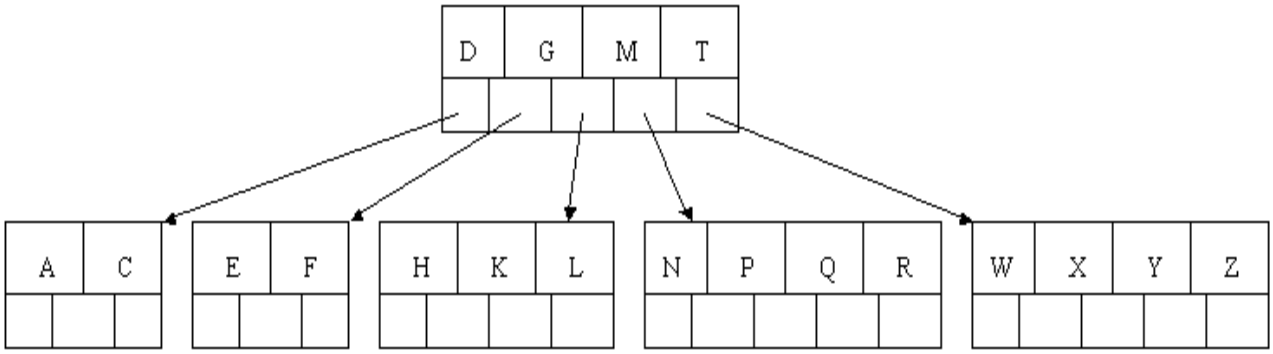
The letters F, W, L, and T are then added without needing any split.



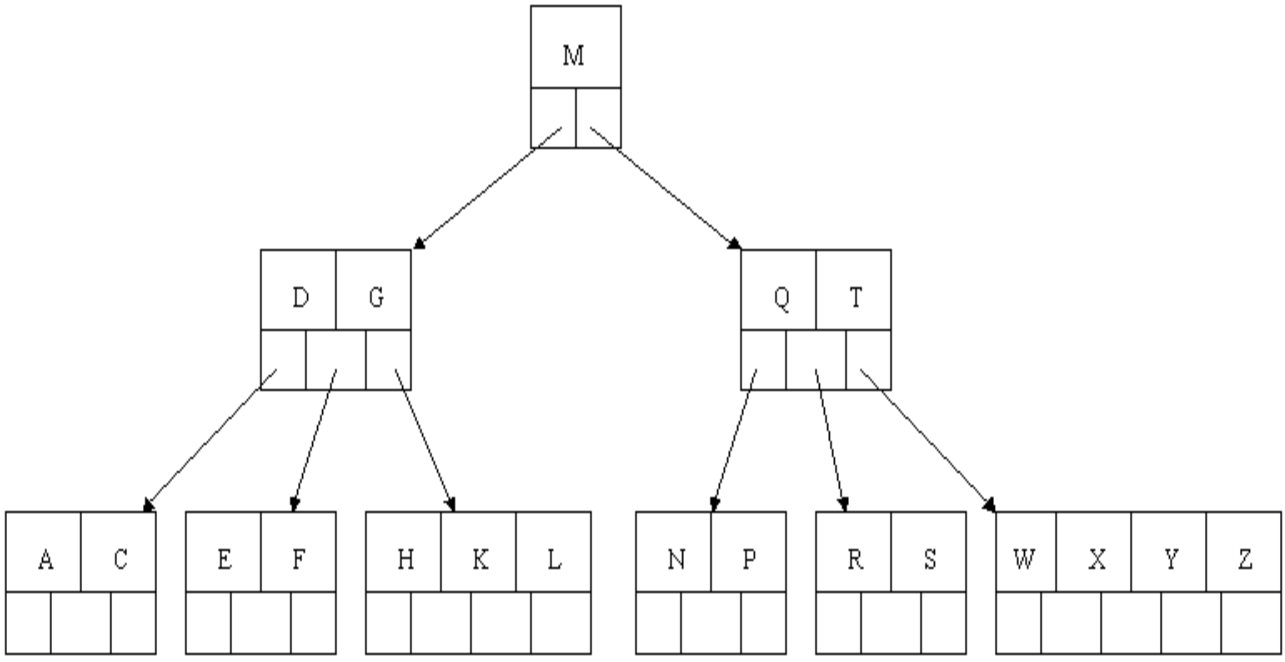
When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes.



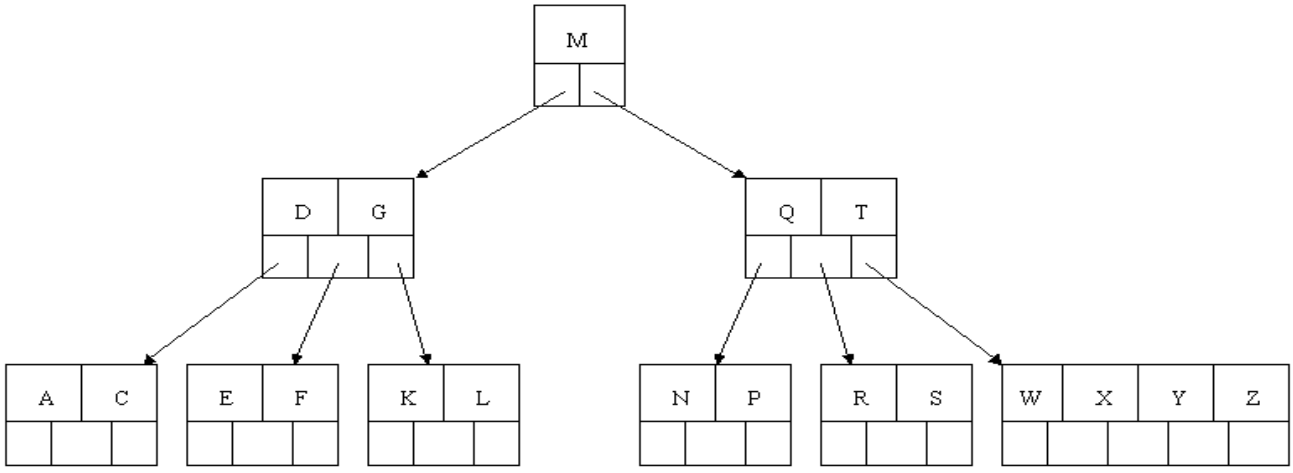
The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting:



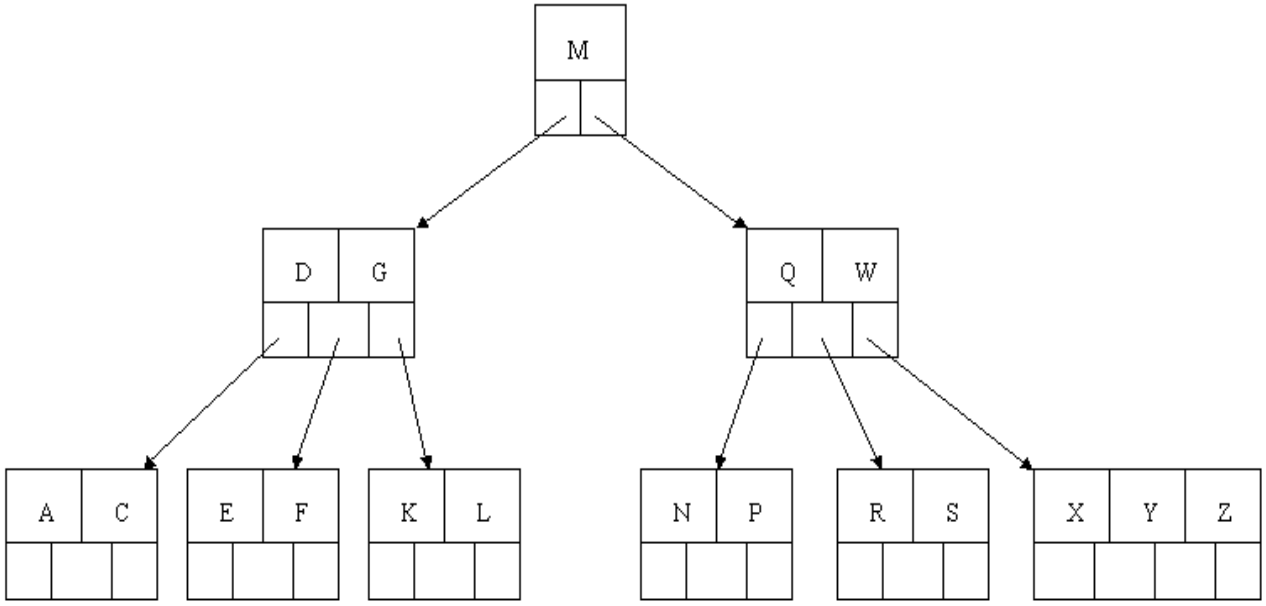
Finally, when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent. However, the parent node is full, so it splits, sending the median M up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains D and G.



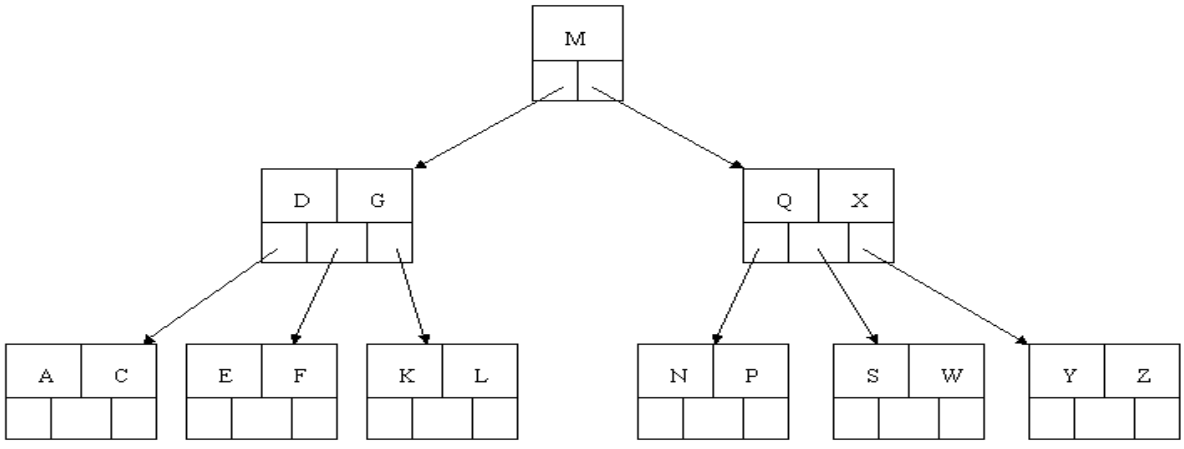
Deleting an Item: In the B-tree as we left it at the end of the last section, delete H. Of course, we first do a lookup to find H. Since H is in a leaf and the leaf has more than the minimum number of keys, this is easy. We move the K over where the H had been and the L over where the K had been. This gives:



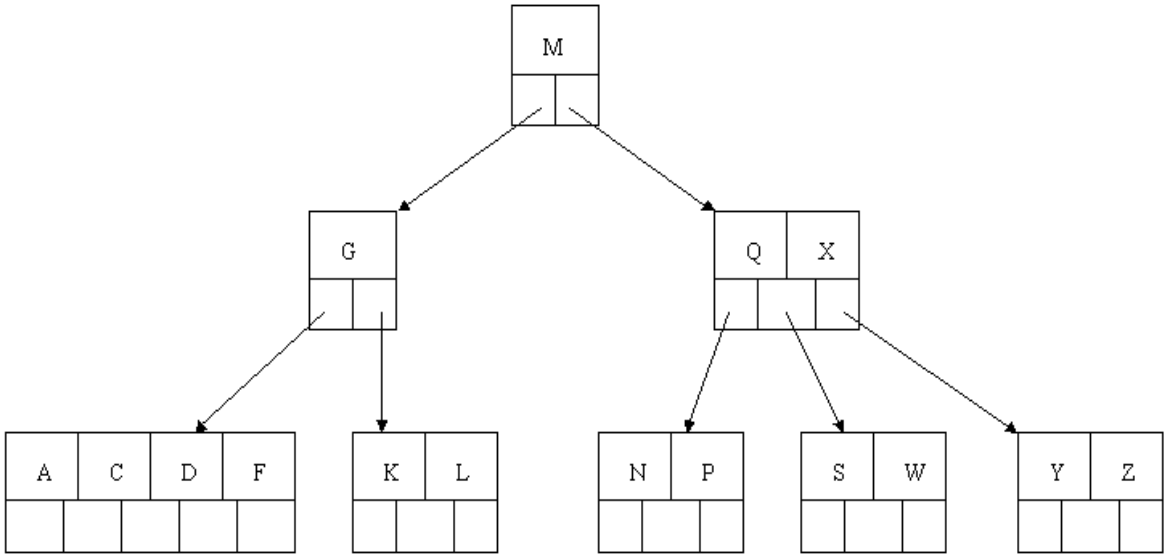
Next, delete the T. Since T is not in a leaf, we find its successor (the next item in ascending order), which happens to be W, and move W up to replace the T. That way, what we really have to do is to delete W from the leaf, which we already know how to do, since this leaf has extra keys. In ALL cases we reduce deletion to a deletion in a leaf, by using this method.



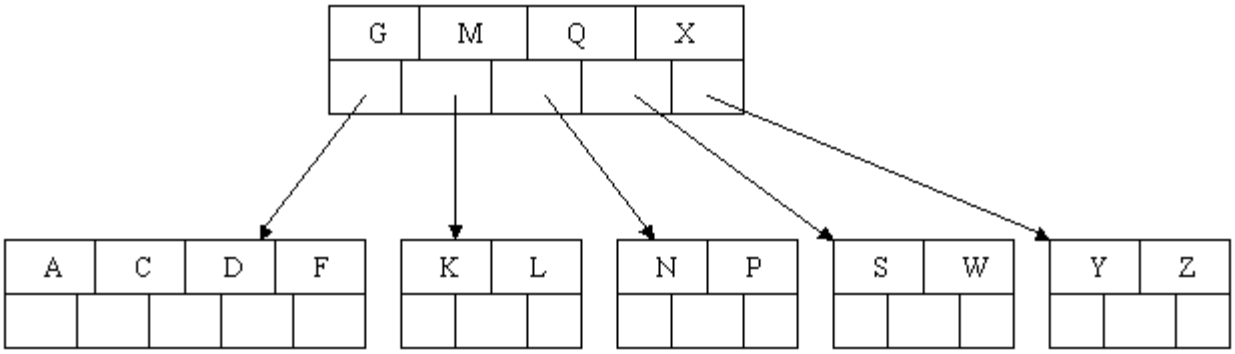
Next, delete R. Although R is in a leaf, this leaf does not have an extra key; the deletion results in a node with only one key, which is not acceptable for a B-tree of order 5. If the sibling node to the immediate left or right has an extra key, we can then borrow a key from the parent and move a key up from this sibling. In our specific case, the sibling to the right has an extra key. So, the successor W of S (the last key in the node where the deletion occurred), is moved down from the parent, and the X is moved up. (Of course, the S is moved over so that the W can be inserted in its proper place.)



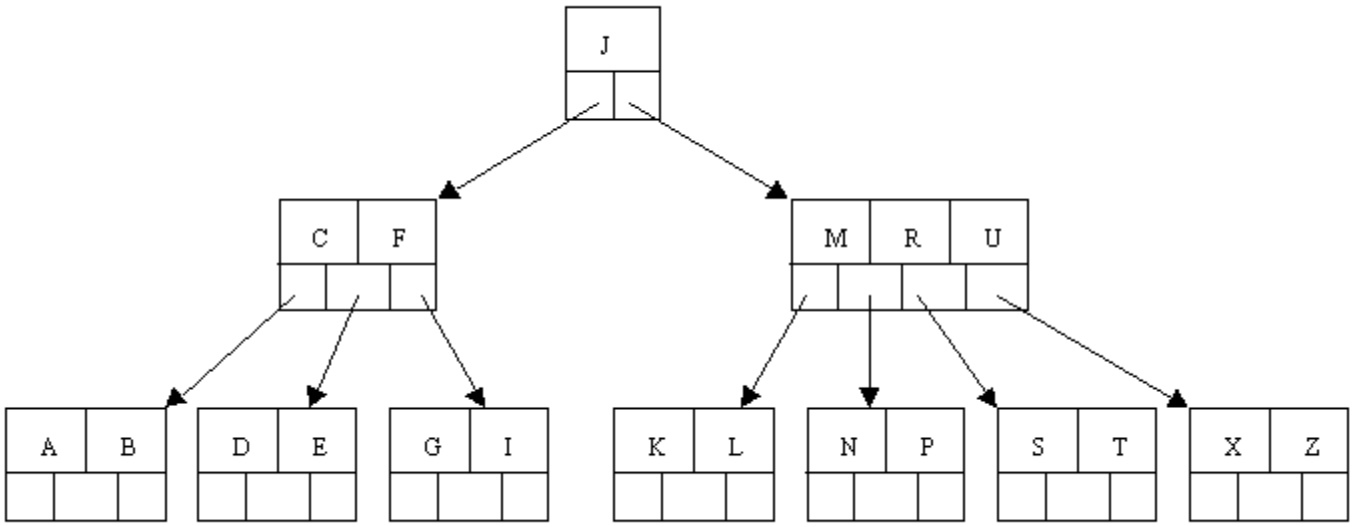
Finally, let's delete E. This one causes lots of problems. Although E is in a leaf, the leaf has no extra keys, nor do the siblings to the immediate right or left. In such a case the leaf has to be combined with one of these two siblings. This includes moving down the parent's key that was between those of these two leaves. In our example, let's combine the leaf containing F with the leaf containing A C. We also move down the D.



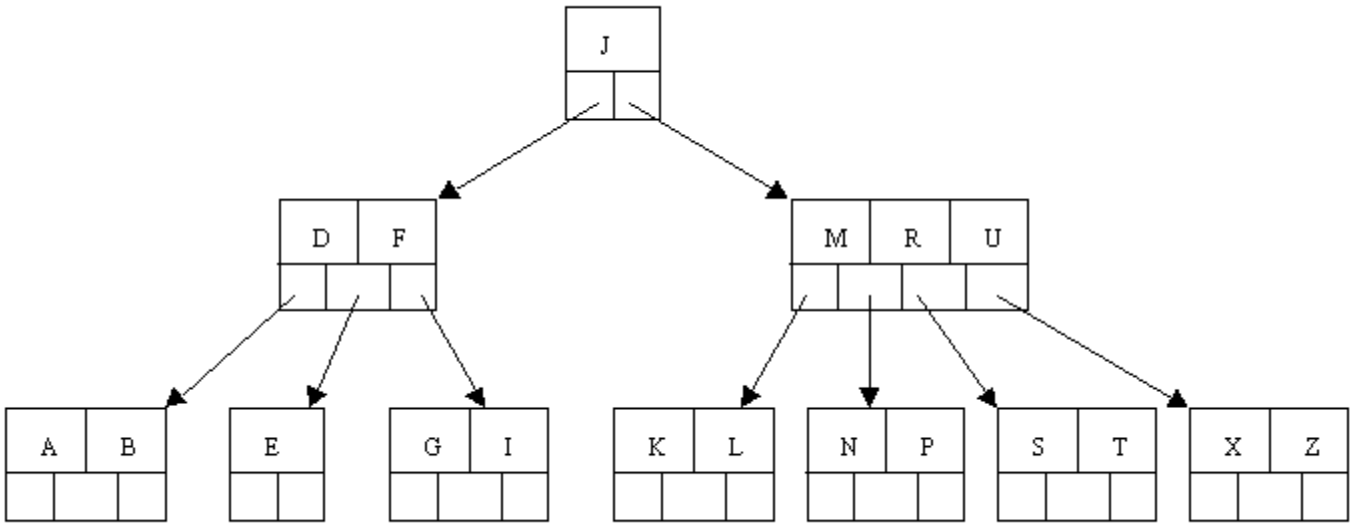
Of course, you immediately see that the parent node now contains only one key, G. This is not acceptable. If this problem node had a sibling to its immediate left or right that had a spare key, then we would again "borrow" a key. Suppose for the moment that the right sibling (the node with Q X) had one more key in it somewhere to the right of Q. We would then move M down to the node with too few keys and move the Q up where the M had been. However, the old left subtree of Q would then have to become the right subtree of M. In other words, the N P node would be attached via the pointer field to the right of M's new location. Since in our example we have no way to borrow a key from a sibling, we must again combine with the sibling, and move down the M from the parent. In this case, the tree shrinks in height by one.



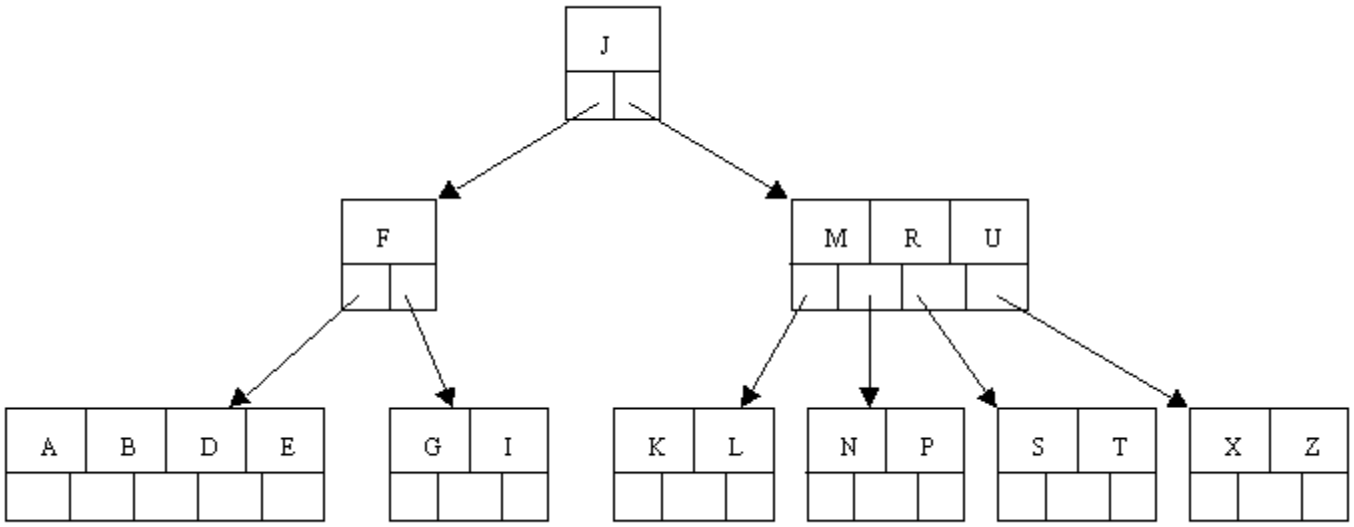
Another Example
Here is a different B-tree of order 5. Let's try to delete C from it.



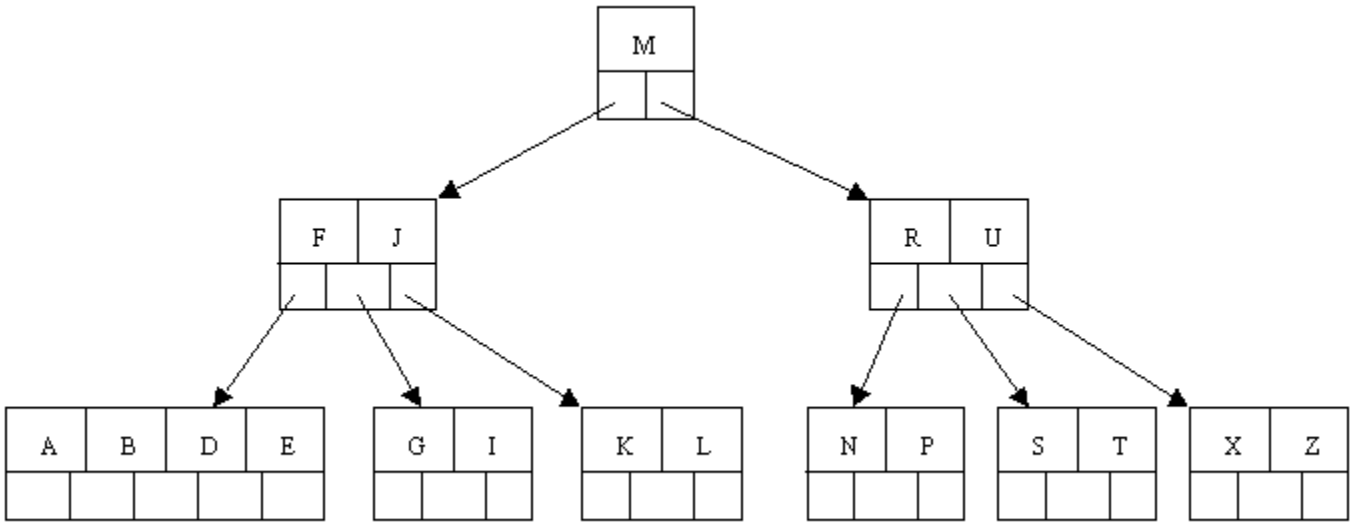
We begin by finding the immediate successor, which would be D, and move the D up to replace the C. However, this leaves us with a node with too few keys.



Since neither the sibling to the left or right of the node containing E has an extra key, we must combine the node with one of these two siblings. Let's consolidate with the A B node.



But now the node containing F does not have enough keys. However, its sibling has an extra key. Thus we borrow the M from the sibling, move it up to the parent, and bring the J down to join the F. Note that the K L node gets reattached to the right of the J.



UNIT-IV

Sorting Techniques: Arranging the data in ascending or descending order are called as sorting techniques. There are many techniques to sort any array.

Insertion sort: In insertion sort from the first element compared with the next of that element. If the previous one is greater than swap the values and so on.

Example: The following table shows the steps for sorting the sequence {3, 7, 4, 9, 5, 2, 6, 1}. In each step, the item under consideration is underlined. The item that was moved (or left in place because it was biggest yet considered) in the previous step is shown in bold.

3 7 4 9 5 2 6 1
3 7 4 9 5 2 6 1
3 7 4 9 5 2 6 1
3 4 7 9 5 2 6 1
3 4 7 9 5 2 6 1
3 4 5 7 9 2 6 1
2 3 4 5 7 9 6 1
2 3 4 5 6 7 9 1
1 2 3 4 5 6 7 9

```
#include <stdio.h>

int main()
{
 int n, array[1000], c, d, t;

 printf("Enter number of elements\n");
 scanf("%d", &n);

 printf("Enter %d integers\n", n);

 for (c = 0; c < n; c++) {
  scanf("%d", &array[c]);
 }

 for (c = 1 ; c <= n - 1; c++) {
  d = c;

  while ( d > 0 && array[d] < array[d-1]) {
   t          = array[d];
   array[d]   = array[d-1];
   array[d-1] = t;

   d--;
  }
 }
```

```
  printf("Sorted list in ascending order:\n");

  for (c = 0; c <= n - 1; c++) {
    printf("%d\n", array[c]);
  }

  return 0;
}
```

selection sort: In this case it's more common to remove the minimum element from the remainder of the list, and then insert it at the end of the values sorted so far. For example :

64 25 12 22 11
11 64 25 12 22
11 12 64 25 22
11 12 22 64 25
11 12 22 25 64

```
#include <stdio.h>

int main()
{
  int array[100], n, c, d, position, swap;

  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);

  for ( c = 0 ; c < n ; c++ )
    scanf("%d", &array[c]);

  for ( c = 0 ; c < ( n - 1 ) ; c++ )
  {
    position = c;

    for ( d = c + 1 ; d < n ; d++ )
    {
      if ( array[position] > array[d] )
        position = d;
    }
    if ( position != c )
    {
      swap = array[c];
      array[c] = array[position];
      array[position] = swap;
    }
  }
```

```
  printf("Sorted list in ascending order:\n");

  for ( c = 0 ; c < n ; c++ )
    printf("%d\n", array[c]);

  return 0;
}
```

Merge sort: If the list is of length 0 or 1, then it is sorted. Otherwise; Divide the unsorted list into two sublists of about half the size; Sort each sublist recursively by re-applying merge sort; Merge the two sublists back into one sorted list.

```
#include <stdio.h>
#include <conio.h>

void main( )
{
   int a[5] = { 11, 2, 9, 13, 57 } ;
   int b[5] = { 25, 17, 1, 90, 3 } ;
   int c[10] ;
   int i, j, k, temp ;
   clrscr( ) ;

   printf ( "Merge sort.\n" ) ;

   printf ( "\nFirst array:\n" ) ;
   for ( i = 0 ; i <= 4 ; i++ )
      printf ( "%d\t", a[i] ) ;

   printf ( "\n\nSecond array:\n" ) ;
   for ( i = 0 ; i <= 4 ; i++ )
      printf ( "%d\t", b[i] ) ;

   for ( i = 0 ; i <= 3 ; i++ )
   {
      for ( j = i + 1 ; j <= 4 ; j++ )
      {
         if ( a[i] > a[j] )
         {
            temp = a[i] ;
            a[i] = a[j] ;
            a[j] = temp ;
         }
         if ( b[i] > b[j] )
         {
```

```
            temp = b[i] ;
            b[i] = b[j] ;
            b[j] = temp ;
         }
      }
   }

   for ( i = j = k = 0 ; i <= 9 ; )
   {
      if ( a[j] <=  b[k] )
         c[i++] = a[j++] ;
      else
         c[i++] = b[k++] ;

      if ( j == 5 || k == 5 )
         break ;
   }
   for ( ; j <= 4 ; )
      c[i++] = a[j++] ;

   for ( ; k <= 4 ; )
      c[i++] = b[k++] ;

   printf ( "\n\nArray after sorting:\n") ;
   for ( i = 0 ; i <= 9 ; i++ )
      printf ( "%d\t", c[i] ) ;

   getch( ) ;
}
```

Searching Techniques: To find out some data from memory is called searching. We can use two types of searching techniques

1. Linear search
2. Binary search

Linear search: In linear search we search data one by one from beginning to end.

```
#include <stdio.h>

int main()
{
  int array[100], search, c, number;

  printf("Enter the number of elements in array\n");
  scanf("%d",&number);

  printf("Enter %d numbers\n", number);

  for ( c = 0 ; c < number ; c++ )
```

```
    scanf("%d",&array[c]);

  printf("Enter the number to search\n");
  scanf("%d",&search);

  for ( c = 0 ; c < number ; c++ )
  {
    if ( array[c] == search )    /* if required element found */
    {
      printf("%d is present at location %d.\n", search, c+1);
            break;
    }
  }
  if ( c == number )
    printf("%d is not present in array.\n", search);

  return 0;
}
```

Binary search: It can only be used for sorted arrays, but it's fast as compared to linear search. If you wish to use binary search on an array which is not sorted then you must sort it using some sorting technique say merge sort and then use binary search algorithm to find the desired element in the list. If the element to be searched is found then its position is printed.

```
#include <stdio.h>

int main()
{
  int c, first, last, middle, n, search, array[100];

  printf("Enter number of elements\n");
  scanf("%d",&n);

  printf("Enter %d integers\n", n);

  for ( c = 0 ; c < n ; c++ )
    scanf("%d",&array[c]);

  printf("Enter value to find\n");
  scanf("%d",&search);

  first = 0;
  last = n - 1;
  middle = (first+last)/2;

  while( first <= last )
  {
    if ( array[middle] < search )
      first = middle + 1;
```

```
    else if ( array[middle] == search )
    {
      printf("%d found at location %d.\n", search, middle+1);
      break;
    }
    else
      last = middle - 1;

    middle = (first + last)/2;
  }
  if ( first > last )
    printf("Not found! %d is not present in the list.\n", search);

  return 0;
}
```

Hashing:

Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value.

Hashing Functions: Several kinds of uniform hashing function are in use.

1. Direct hashing, the key is the address without any algorithmic manipulation. The data structure must therefore contain an element for every possible key. While the situations where direct hashing are limited, when it can be used it is very powerful because it guarantees that there are no collisions.
   Limitations: Large key value.

   a) Mid-Square (middle of Square)
   9452 * 9452 = 89340304 = 3403
   As a variation on the mid-square method, we can select a portion of the key, such as the middle three digits, and then use them rather than the whole key. This allows the method to be used when the key is too large to square.
   379452: 379 * 379 = 143641 = 364
   121267: 121 * 121 = 014641 = 464

   b) Modulo-Division Also known as Division-remainder.
   Address = Key MOD Table size
   While this algorithm works with any table size, a list size that is a prime number produces fewer collisions than other list sizes.

   c) Folding:There are two folding methods that are used, fold shift and fold boundary. In fold shift, the key value is divided into parts whose size matches the size of the required address. Then the left and right parts are shifted and added with the middle part. In fold boundary, the left and right numbers are folded on a fixed boundary between them and the center number.

a. Fold Shift

Key: 123456789

123
456
789

1368 (1 is discarded)

b. Fold Boundary

Key: 123456789

321 (digit reversed)
456
987 (digit reversed)

1764 (1 is discarded)

d) Digit-Extraction

Using digit extraction, selected digits are extracted from the key and used as the address. For example, using a six-digit employee number to hash to a three-digit address(000999), we could select the first, third. and fourth digits (from left) and use them as the address.

379452 = 394
121267 = 112

Non-Numeric Keys

If the identifiers were restricted to be at most six characters long with the first one being a letter and the remaining either letters or decimal digits, then there would be

$$T = \sum_{0<=i<=5} (26 * 36^i) > 1.6 * 10^9.$$

Collision Resolution

With the exception of the direct method, none of the methods used for hashing are one-to-one mapping. This means that when we hash a new key to an address, we may create a collision. There are several methods for handling collisions, each of them independent of the hashing algorithm. Before we discuss the collision resolution methods, we need to cover few basic concepts:

a) Load Factor

The load Factor alpha of a hash table of size M with N occupied entries is defined by

alpha = N/M

b) Clustering

Some hashing algorithms tend to cause data to group within the list. This tendency of data to build up enevenly across a hashed table is known as clustering.

1. Primary Clustering: Primary clustering occurs when data become clustered
around a home address.

2. Secondary Clustering: Secondary clustering occurs when data become grouped along a collision path throughout the list.

Open Addressing
The first collision resolution method, open addressing, resolves collisions in the home area. When a collision occurs, the home area addresses are searched for an open or unoccupied element where the new data can be placed.
Examples of Open Addressing Methods:

a) Linear Probe
i = H(key) is the home address. If it is available we store the record, otherwise, we increase i by k, i = (i + k) mod M (k = 1, 2, 3, ...).

Linear probing gives rise to a phenomenon called primary clustering.

b) Quadratic Probe
If there is a collision at hash address h, this method probes the table at locations h+1, h+4, h+9, ..., that is, at locations h + i^2 (mod table size) for i = 1, 2, ... That is, the increment function is i^2. Quadratic probing substantially reduces clustering, but it is not obvious that it will probe all locations in the table, and in fact it does not. For some values of hash_size the function will probe relatively few position in the table.

c) Double Hashing
Double Hashing uses nonlinear probing by computing different probe increments for different keys.

It uses two functions. The first function computes the original address, if the slot is available (or the record is found) we stop there, otherwise, we apply the second hashing function to compute the step value.

i = H1 (key) to compute the home addressh2(key) = step value = Max (1, Key DIV M)MOD M

i = i + step value we repeat this until we find a place or we find the record.
Double hashing avoids primary and secondary clustering

d) Chaining

One way of resolving collisions is to maintain M linked lists, one for each possible address in the hash table. A key K hashes to an address i = h(k) in the table. At address i, we find the head of a list containing all records having keys that have hashed to i. This list is then searched for a record containing key K.

References:

*Data structures using C "**yashwant kanetkar**"*
*Data structures "**Aaron M. Tenenbaum**"*
*Data structures using C" **Anuradha**"*
en.wikipedia.org
ww2.valdosta.edu
en.wikibooks.org
www.princeton.edu
www.i-programmer.info
www.programmingsimplified.com
www.programmingsimplified.com
www.cquestions.com
www.cprogramming.com
www.Gurukpo.com
xlinux.nist.gov
www.cs.indiana.edu
www.informatik.uni-freiburg.de