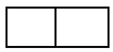
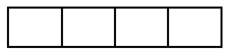
DYNAMIC MEMORY ALLOCATION

MEMORY ALLOCATION OPERATOR NEW

```
T  *p; //declare p as a pointer
 p= new T // p is the address of Memory for data type T

int *ptr1; // size of int is 2
long *ptr2; // size of long in 4

ptr1= new int;
ptr2= new long;
```

ptr1→  ptr2→ 

by default, the content in memory have no initial value. If such a value is desired, it must be supplied as a parameter when the operator is used:

```
p=new T(value);

ptr2= new long(1000000)
```

DYNAMIC ARRAY ALLOCATION

```
p=new T [n]; // allocate an array of n items of type T
```

Example:

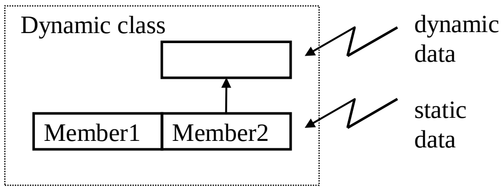
```
 long *p;
p=new long [50] // allocate an array of 50 long integers
if (p==NULL)
{
cerr<< "Memory allocation error!<< endl;
exit(1); // terminate the program
}
```

THE MEMORY DEALLOCATION OPERATOR DELETE

```
T *p, *q;     // p and q are pointers to type T
p=new T;     // points to a single item
q new T[n];  //points to an array of elements

delete p; // deallocates the pariable pointed by p
delete [ ] q; // deallocates the entire array pointed  by q
```

ALLOCATION OF OBJECT DATA

Example:



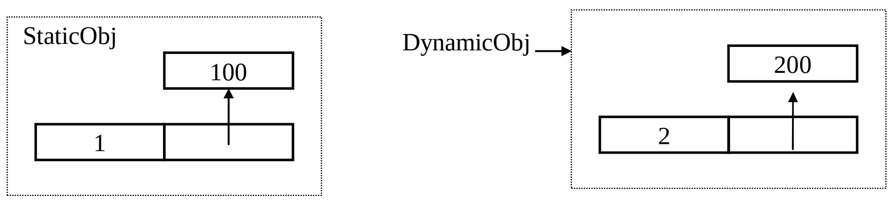

```
template <class T>
class DynamicClass
{ private:
// variable of type T and a pointer to data of type T
T  member1;
T *member2

public:
//constructor with parameters to initialize member data
DynamicClass(const T  &m1, const  T &m2)
// copy constructor : create a copy of the input object
DynamicClass(const DynamicClass<T> & obj)
// some methods…
….
//assignment operator
DynamicClass<T> *operator=(cont DynamicClass<T> *rhs)
//destructor
DynamicClass(void)
}

// class implementation
//constructor with parameters to initialize member data
DynamicClass<T>::DynamicClass(const T  &m1, const  T &m2)
{
// parameter m1 initializes static member
member1=m1;
//allocate dynamic memory and initialize it with value m2
member2=new T(m2)
cout << "Constructor:"<<member1<<'/'<<*member2<<endl;
```

Example: The following statements define a static variable staticObj.The static Obj has parameters 1 and 100 that initialize the data members:

//Dynamic Class object
DynamicClass<int> staticObj(1,100)

In the following, the object dynamicObj points is created by the new operator. Parameters 2 and 200 are supplied as parameters to the constructor:

//pointer variable
DynamicClass<int> *dynamicObj;
//allocate an object
dynamicObj=new dynamicClass<int>(2,200)



Runinng the program results in

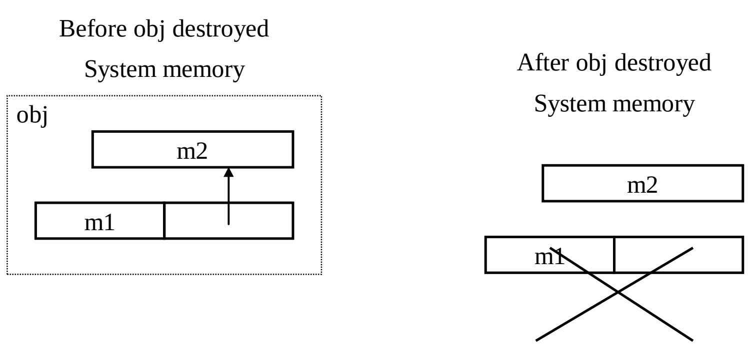
Constructor: 1/100
Constructor: 2/200

DEALLOCATION OBJECT DATA: THE DESTRUCTOR

Consider the function DestroyDemo that creates a DynamicClass object having integer data

void DestroyDemo(int m1,int m2)
{DynamicClass<int> obj(m1,m2);
}

Upon return from DestroyDemo obj is destroyed; however the process does not deallocate the dynamic memory associated with the object:



Dynamic data still remains in the system memory. For effective memory management, we need to deallocate the dynamic data within the object at the same time the object being destroyed. We need to reverse the action of the constructor, which originally allocated the dynamic data. The C++ language provides a member function, called the destructor, which is called automatically when an object is destroyed. For DynamicClass, the destructor has the declaration:

```
~DynamicClass(void);
```

The character "~" represents "complement", so ~DynamicClass is the complement of the constructor DynamicClass. A destructor never has a parameter or a return type. For our sample class, the destructor is responsible to deallocate the dynamic data for member2.

```
// destructor: deallocates memory allocated by the constructor
template <class T>
DynamicClass<T>:~DynamicClass(void);
{cout<<"Destructor:"<<member1"<<'/'<<member2<<endl;
 delete member2;
}
```

The destructor is called whenever an object is deleted. When a program terminates, all global objects or objects declared in the main program are destroyed. For local objects created within a block, the destructor is called when the program exits the block.

Example

```
void DestroyDemo(int m1, int m2)
{DynamicClass<int> Obj(m1,m2) ←————————          Constructor for Obj(3,300)
}  ←——————————————————————————————————          Destructor for Obj
void main(void)
{DynamicClass<int> Obj1(1,100), *Obj2; ←——      Constructor for Obj1(1,100)
  Obj2=new DynamicClass<int>(2,200);   ←——      Constructor for *Obj2(2,200)
  DestroyDemo(3,300);
  delete Obj2;  ←—————————————————————————      Destructor for Obj2
}  ←——————————————————————————————————          Destructor for Obj2
```

running the program results in the output:

```
Constructor: 1/100
Constructor: 2/200
Constructor: 3/300
Destructor: 2/200
Destructor: 1/100
```

ASSIGNMENT AND INITIALIZATION

Assignment and initialization are basic operation that apply to any object. The assignment Y=X causes a bitwisecopy of the data from object X to the data in object Y. Initialization creates a new object that is a copy of another object. The operations are illustrated with objects X and Y.

```
// initialization
DynamicClass X(20,50), Y=X;
//creates DynamicClass objects X and Y
// data in Y is initialized by data in X


// assignment
Y=X;
//data in Y is overwritten by data in X
```

Special consideration must be used with dynamic memory so that unintended errors are not created. We must create new methods that handle object assignment and initialization.

ASSIGNMENT ISSUES

The assignment statement of B=A causes the data in A to be copied to B

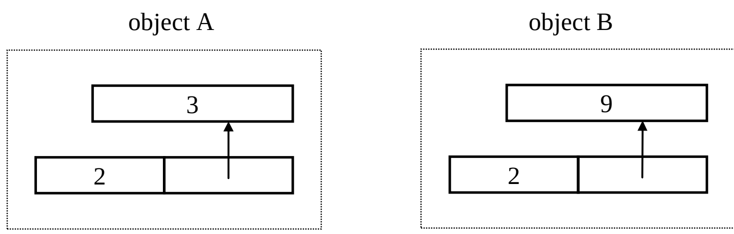
```
member1 of  B=member1 of A //copies static data from A to B
member2 of B=member2 of A  //copies pointer from A to B
```

Example

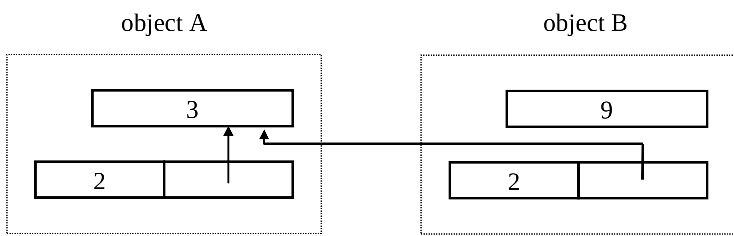
```
void F(void)
{DynamicClass<int> A(2,3), B(7,9);
B=A
}
```

After execution of

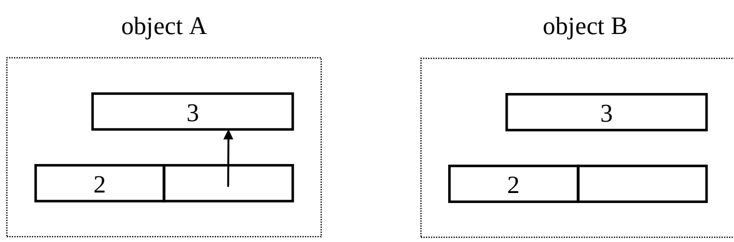
```
DynamicClass<int> A(2,3), B(7,9);
```



After execution of B=A we have



although it was desired



Solution is Overloading the assignment operator

```
// Overloaded assignment operator = returns a reference to the current object
template <class T>
DynamicClass<T>& operator= (const DynamicClass <T>& rhs)
{//copy static data member from rhs to the current object
member1=rhs.member1
// content of the dynamic memory must be same as that rhs
*member2=*rhs.member2;
cout <<"Assignment Operator: "<<member1<<'/'<<*member2<<endl
return *this;
 //reserved word this is used to return a reference to the current object
}
void main (void)
{ DynamicClass <int> A(2,3), B(7,9);
B=A;   // ≡ B.operator = (A)
}
```

INITIALIZATION ISSUES

Object initialization is an operation that creates a new object that is a copy of another object. Like assignment, when the object has dynamic data, the operation requires a specific member function,called the copy constructor.

```
DynamicClass<int> A(3,5), B=A; //initialize object B with A
```

The declaration creates object a whose initial data are member1=3 and *member2=5. The declaration of B creates an object with two data members that are then structured to store the same data values found in A.

In addition to performing initialization when declaring objects, initialization also occurs when passing an object as a value parameter in a function. For instance, assume function F has a value parameter X of type DynamicClass<int>.

```
DynamicClass<int> F(DynamicClass<int> X) // value parameter
{DynamicClass<int> obj;
…..
return obj
}
```

When calling block uses object A as the actual parameter, the local object X is created by copying A:

```
DynamicClass<int> A(3,5), B(0,0);    //declare objects
B=F(A)                               //call F by copying A to X
```

When the return is made from F, a copy of obj is made, the destructor for the local object X and obj are called, and the copy of obj is returned as the value of the function

CREATING A COPY CONSTRUCTOR

In order to properly handle classes that allocate dynamic memory, C++ provides the copy constructor to allocate dynamic memory for the new object and initialize its data values

The copy constructor is a member function that is declared with the class name and a single parameter. Because it is a constructor, it does not have a return value

```
//copy constructor: initialize new object to have the same data as obj.
template <class T>
DynamicClass<T>:: DynamicClass(const DynamicClass<T& obj)
{// copy static data member from obj to current object
member1=obj.member1;
//allocate dynamic memory and initialize it with value *obj.member2
```

```
member2=new T(*.member2);
cout<<"Copy Constructor:"<<member1<<'/'<<member2<<endl;
}
```

If a class has a copy constructor, it is used by the compiler whenever it needs to perform initialization. The copy constructor is used only when an object is created

Despite their similarity, assignment and initialization are clearly different operations. Assignment is done when the object on the left-handside already exeists. In the case of initialization, a new object is created by copying data from an existing object.

The parameter in a copy constructor <u>must be passed by reference</u>. The consequence of failing to do so may result in catastrophic effects if it is not recognized by the compiler. Assume we declare the copy constructor

```
DynamicClass(DynamicClass<T> X)
```

The copy constructor is called whenever a function parameter is specified as call by value. In the copy constructor, assume object A is passed to the parameter X by value

```
DynamicClass(DynamicClass     X)
                         ↑
                         A
```

Since we pass A to X by value, the copy constructor must be called to handle the copying of A to X. This call in turn needs the copy constructor, and we have an infinite chain of copy constructor calls. Fortunately, this potential trouble is caught by the compiler, which specifies that the parameter must be passed by reference. In additiion, the reference parameter X should be declared constant, since we certainly do not want to modify the object we are copying.

```
# include <iostream.h>
# include "dynamic.h"
template <class T>
DynamicClass<int> Demo(DynamicClass<T> one, DynamicClass& two, T m)
{ DynamicClass<T> obj(m,m);
  return obj;
}
void main()
{ DynamicClass<int> A(3,5), B=A, C(0,0);
  C=Demo(A,B,5);
}
```

Running the program results in

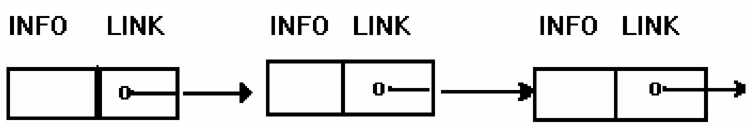
```
Constructor: 3/5                // construct A
Copy Constructor: 3/5           // construct B
Constructor: 0/0                // construct C
Copy Constructor: 3/5           // construct one
Constructor: 5/5                // construct obj
Copy Constructor: 5/5           // construct return object for Demo
Destructor: 5/5                 // destruct obj  upon return
Destructor: 3/5                 //  destruct one
Assignment Operator: 5/5        // assign return object of Demo to C
Destructor: 5/5                 // destruct return  object of demo
Destructor: 5/5                 // destruct C
Destructor: 3/5                 // destruct B
Destructor: 3/5                 // destruct A
```

LINKED LISTS

Arrays are not efficient in dealing with problems such as:

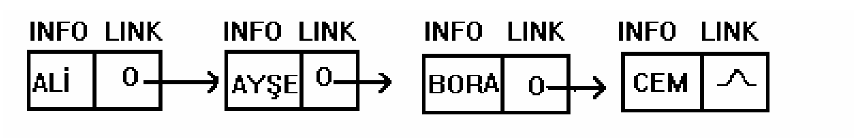
. Joining two arrays,
. Insert an element at an arbitrary location.
. Delete an element from an arbitrary location

To overcome these problems, another data structure called linked list can be used in programs.



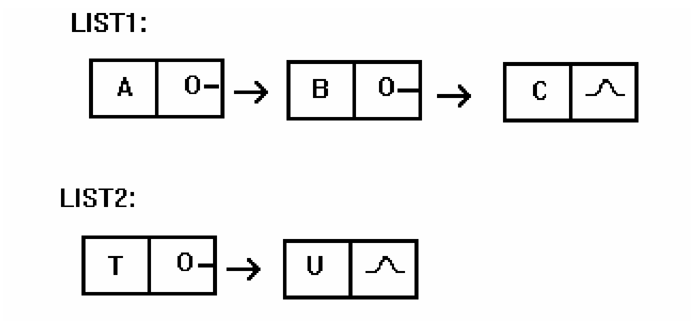
Linked list is formed of a set of data items connected by link fields (pointers). So, each node contains: a) an info (data) part, b) a link (pointer) part
* Nodes do not have to follow each other physically in memory
*The linked list ends with a node which has "^" (nil) in the link part,showing that it is the last element of the chain.

Example:

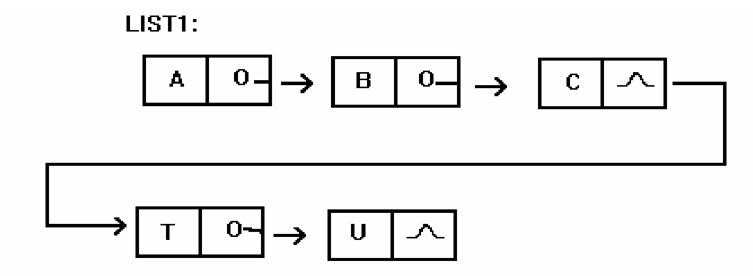


The physical ordering of this linked list in the memory may be

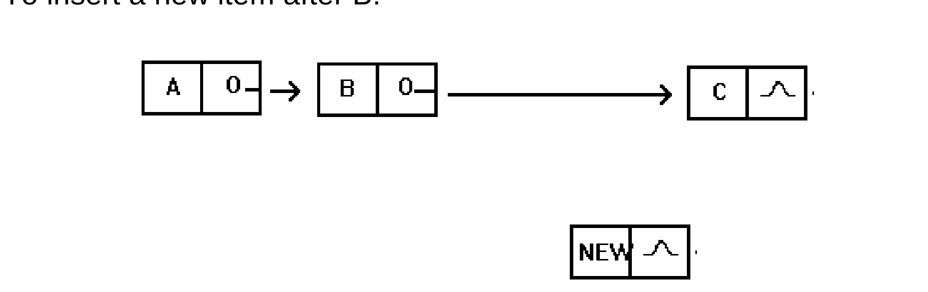
ADDR	MEMORY	
1007	ALİ	1063
	...	...
1024	CEM	^
	...	...
1062	BORA	1024
1063	AYŞE	1062



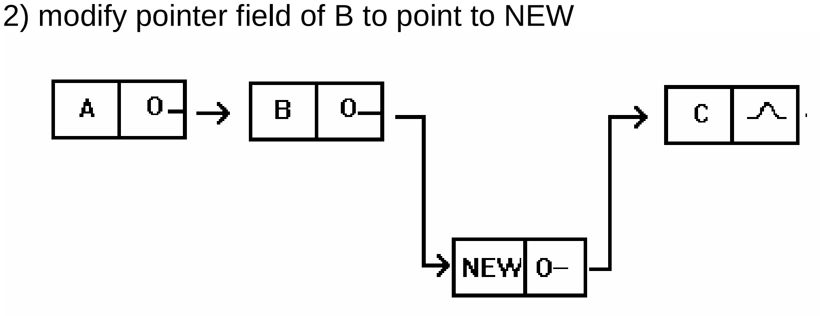
To join LIST1, LIST2: modify pointer of "C" to point to "T".



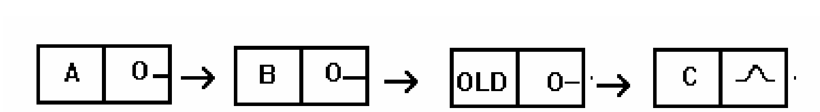
To insert a new item after B:



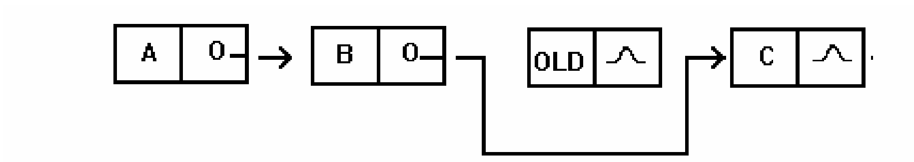
1) modify pointer field of NEW to point to C
2) modify pointer field of B to point to NEW



To delete an item coming after B,

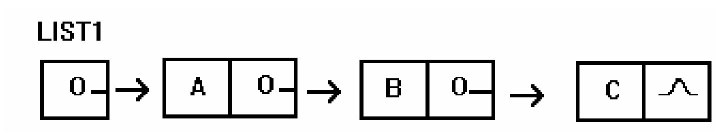


1) modify pointer field of B, to point to the node pointed by pointer of OLD
2) modify pointer field of OLD as ^ (not to cause problem later on)



Some Problems with linked lists can be listed as follows:
1) They take up extra space because of pointer fields.
2)To reach the n'th element, we have to follow the pointers of (n-1) elements sequentially. So, we can't reach the n'th element directly.

For each list, let's use an element "list head" which is simply a pointer pointing at the first entry of the list:



Implementation Node and Linked List Classes in C++

```
//declaration of Node Class
template <class T>
class Node
{private:
Node <T> *next; // next  part is a pointer to nodes of this type
public :
T data; // data part is public
// constructor
Node (const T &item, Node<T>*  ptrNext=0);
//list modification methods
void InsertAfter(Node<T> *p);
Node <T> *DeleteAfter(void);
//get address of next node
Node<T> *NextNode(void) const;
}
```

Note: that the pointer member is private while the data member is public. To avoid the need for function *NextNode, we could declare *next to be public.

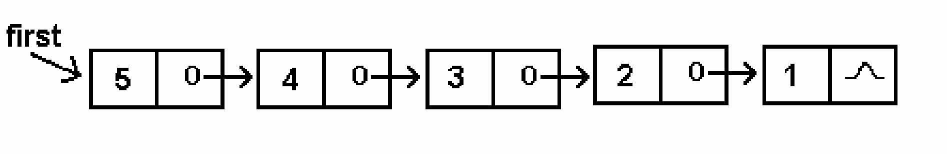
```
// Class Implementation
//constructor
template <class T>
Node <T>::Node(const T& item, Node<T>* ptrnext): data(item), next(ptrnext)
{}

//access nextptr
```

```
template <class T>
Node <T> *Node<T>::NextNode(void) const
{return next;
}

//insert node pointed by p after the current one
template <class>
void Node<T>::InsertAfter(Node<T> *p)
{ // syntax       p→next ≡ *p.next
  p->next=next;
  next=p; //also note correct sequence of operation
}

//delete node following the current node and return its address
Node<T> *Node<T>::DeleteAfter(void)
{
//save address of node to be deleted
Node <T> *tempPtr=next;
//if no successor, return NULL
if (next==NULL)
        return NULL;
//delete next node by copying its nextptr to the
//nextptr of current node
        next=tempPtr->next;
//return pointer to deleted node
        return tempPtr;
}
```

Now, let's define a template-function GetNode that dynamically allocates a node and initializes it

```
template <class T>
Node<T> *GetNode(const T& item, Node<T> *nextPtr=NULL)
{
   Node<T> *newNode; //declare pointer
   newNode=new  Node<T>(item, nextPtr);
//allocate memory and pass item and nextptr to the constructor which creates the object
//terminate program if allocation not successful
            if (newNode==NULL)
{cerr<<"Memory allocation failed"<<endl;
            exit(1);
return newNode;
}
```

```
Node <int> *first=null;
for (i=1; i<=5; i++)
  first = getNode(I, first);
```




```
//function to insert a new item at the front of a list
template <class T>
void InsertFront(Node<T> &head, T item)
//we are passing in the address of the head pointer by &head so that it can be modified)
{
//allocate new node so that it points to the first item in the original list,
// and updated  head pointer to point to the new node
            head=GetNode(item,head);
}
```

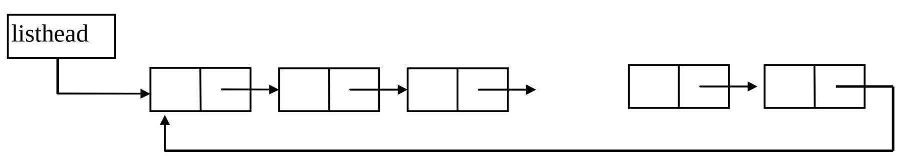
execises:
1) write a function to insert a new item at the end of a list
2) write a function to find the first occurrence of "key" in a key and delete it
3) all occurences
4) write a function to reverse the order of a list
5) convert a linear list to a circular list
6) convert a circular list to a linear list

Example: Function to delete the first occurrence of "key" in a list

```
template <class T>
void Delete(Node <T>* &head, T key)
{Node<T> *currPtr=head, *prevPtr=NULL
//return if listempty
if (currPtr==NULL)
return;
while(currPtr !=NULL&&curPtr->data!=key)
{
    prevPtr=currPtr; //keep prev item to delete next
    currPtr=currPtr->NextNode();
}
if (currPtr!=NULL) //i.e. keyfound
{if (prevPtr==NULL) //i.e key found at first entry
          head=head->NextNode();
else
          prevPtr->DeleteAfter();
          delete currPtr; //remove memory space to memory manager
  }
}
```

Circular Lists

The last node points to the first



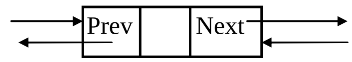
Whereas returning a whole list to lavs (list of available space) takes O(n) operations with a linear list,

```
temp=lavs;
lavs=listehead->next
listhead->.next=temp;
listhead=nil;
```

this takes O(1) time

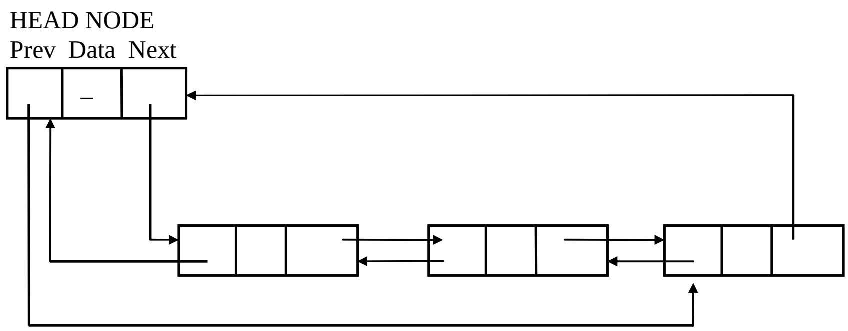
Doubly Linked Lists

1) Easy to traverse both ways



2) Easy to delete the node which is pointed at (rather than the one following it, as in the case of simply linked lists)

Example: A doubly linked cicular list:

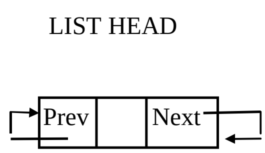


Conventions:

*HEAD NODE: does not carry data, it simply points to the first node (NEXT) and the last node (PREV)

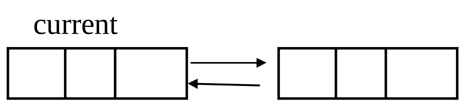
* NEXT pointer of the last node & the PREV pointer of the first node point to the HEAD NODE

So empty List



Insertion into a doubly linked list:

```
void  dinsert(Node *p,  q)
/*insert node p to the right of current node*/
{
        p->next=q;
        p->prev=this;
        p->next->prev=p;
        q->next=p;
};
```



Exercises

Write a procedure Add(Node <T> * p1, *p2) that will add/multiply two polynomials represented by doubly linked lists whose head nodes are pointed by P1 & P2

Other linked list examples

1. consider $p(x,y,z)=2xy^2z^3+3x^2yz^2+4xy^3z+5xy^3z+5x^2y^2$
   rewrite so that terms are ordered lexicographically , that is x in ascending order, for erqual x powers y in a.o., then z in a.o

2. Write a procedure to count the no. of nodes in a one way linked list
3. Search for info 'x' and delete that node if it is found (one way)
4. Reverse the direction of the links in a one way linked circular list

Implementation of a linked list as a class

Note: all list processing functions can be implemented using only the Node class and node operations, however this makes it more object oriented.

```
# include <iostream.h>
#include <stdlib.h>
#include "node.h"
//assuming that "node.h" contains a complete definition of the node class
template <class T>
class LinkedList
{Private:
//pointers to access front and rear
Node <T>  *front, *rear;
// pointers for traversal
Node <T> *PrevPtr, currPtr;
//count of elements in list
int size;
//relative position of current
int position;
//private methods to allocate and deallocate nodes Node<T> *GetNode(const T&
item, Node<T>* ptrNext=NULL);
void FreeNode(Node<T> *p);
// copy list L to current list
void CopyList(const LinkedList<T>& L);
public:
//constructor
LinkedList(void);
LinkedList(const LinkedList<T>& L);
// Destructor
~LinkedList(void)
// assignment
LinkedList<T>&operator=(const LinkedList<T>&L);
//check list status
int ListSize(void) const;
```

```
int List Empty(void) const;
//Traversal
void Reset (int pos=0);
//sets prevPtr to currPtr and currPtr to the address corresponding to given pos
// if pos==0 the method sets prevPtr to currPtr and currPtr to the front of the list
void Next(void); // advance both pointers by one node
int EndOfList(void) const; // indicate whether currPtr is pointing to the last node
int CurrentPosition(void) const; //returns current location so that it can be stored and given to Reset for Later processing
// Insertion methods
void InsertFront(const T& item); // i.e. newNode=GetNode(item); front=newNode;
void InsertRear(constT& item);
void InsetAt(const T& item); // after the node currently pointed by prevptr
void InsertAfter(const T& item); // i.e. after the node currently pointed by currPtr
//Deletion
T Deletefront(void);
void DeleteAt(void);
//Data retrieval and/or modification
T& Data(void); //note that the reference to the data item is returned
// e.g. L.data()=L.Data()+8;
void clearList(void); //remove allnodes and  mark list as empty
} // end of class linked List
```

Eg. To scan and process whole list L:

```
for (L.Rest();!L.EndOfList();L(Next())
{//process current location}
```

Example: print the content of a list

```
void PrintList (const LinkedList<T> &L)
      L.Reset()
      if (L.ListEmpty())
          cout<<"EmptyList" \n";
else
     while (!L.EndOfList())
     {    cout<<L.Data()<<endl;
          L.Next
      }
```

Exercise: Implement all methods of LinkedList

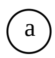
TREES

A tree is a set of nodes which is either null or with one node designated as the tree and the remaining nodes partitioned into smaller trees, called sub-trees.
Example:

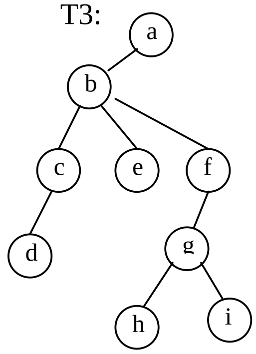
T1={} (NULL Tree)
T2={a} a is the root, the rest is T1
T3={a, {b,{c,{d}},{e},{f,{g,{h},{i}}}}

graphical representation:

T2:



T3:

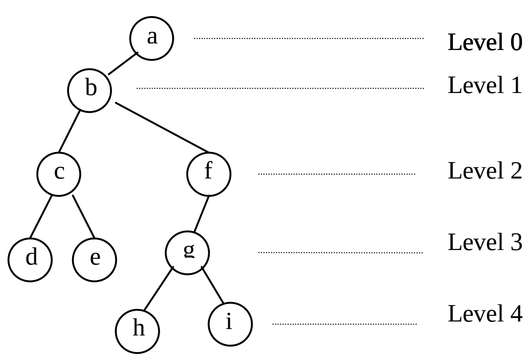


- The <u>level</u> of a node is the length of the path from the root to that node
- The <u>depth</u> of a tree is the maximum level of any node of any node in the tree
- the <u>degree</u> of a node is the number of partitions in the subtree which has that node as the root
- nodes with degree=0 are called <u>leaves</u>

BINARY TREE

Binary tree is a tree in which the maximum degree of any node is 2.

e.g.



- A binary tree may contain up to 2^n nodes at level n.

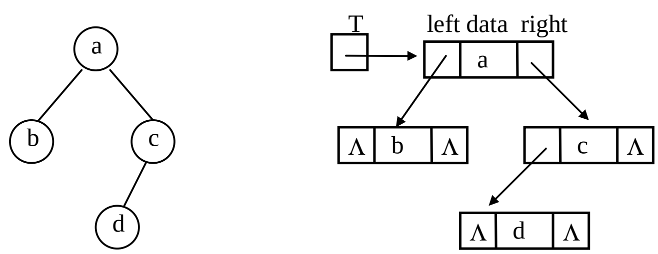
- A complete binary tree of depth N has 2k nodes at levels k=0,…,N-1 and all leaf nodes at level N occupy leftmost positions.

- If level N has 2N nodes as well, then the complete binary tree is a <u>full</u> tree.
- If all nodes have degree=1, the tree is a degenerate tree (or simply linked list)

e.g. a degenerate tree of depth 5 has 6 nodes
a full tree of depth 3 has 1+2+4+8=15 nodes
a full tree of depth N has $2^{N+1}-1$ nodes
a complete tree of depth N has $2^N-1<m\leq 2^{N+1}-1$ nodes

exercise: what is the depth of a complete tree with m nodes?

DATA STRUCTURES AND REPRESENTATIONS OF TREES



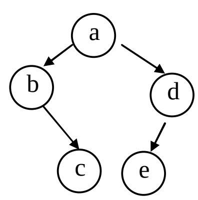

```
template <class T>
class  TreeNode
{private:
TreeNode<T> *left;
TreeNode<T>  *right;
public:
T data;
//constructor
        TreeNode(const T &item, TreeNode<T> *lptr=NULL, TreeNode<T>
                        *rptr=NULL);
//access methods for the pointer fields
        TreeNode<T>* Left(void) const;
        TreeNode<T>* Right(void) const;
};
```

```
//constructor
template <class T>
TreeNode<T>:: TreeNode(const T &item, TreeNode<T> *lptr,
                                    TreeNode<T>    *rptr):    data(item),    left(lptr),
right(rptr)
{}
```

```
//a function dynamically allocate memory for a new object
template <class T>
Treenode<T> *GetTreeNode(T item, TreeNode<T> *lptr=NULL, TreeNode<T>
*rptr=NULL)
{TreeNode<T> *p;
p=new TreeNode<T> (item, lptr, rptr);
if (p==NULL) // if "new" was unsuccessful
{cerr<<éMemory allocation failure"<<endl;
exit(1);
}
return p;
}
```

```
Example TreeNode<char> *t;
t=GetTreeNode('a', GetTreeNode('b',NULL, GetTreeNode('c')),
                    GetTreeNode('d', GetTreeNode('e')));
```

result:




```
// a function to deallocate memory template <class T>
void FreeTreeNode(TreeNode <T> *p)
{delete p;}
```

Tree Traversal Algorithms:

1: DEPTH-FIRST:

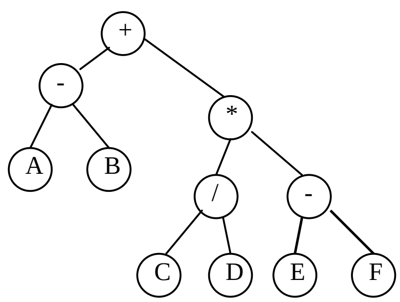
Inorder: 1. Traverse left subtree
2. Visit node (.e. process node)
3. Traverse right subtree

Preorder: 1.Visit node
2. Traverse Left
3. Traverse right

Post-order: 1. Traverse left
2. Traverse right
3. Visit node

Example:

An arithmetic expression tree stores operands in leafs, operators in non-leaf nodes:



inorder traversal: (LNR)

(A-B)+((C/D)*(E-F))
A-B+C/D*E-F (paranthesis assumed)

postorder traversal: (LRN)

AB-C/EF-*+

preorder traversal: (NLR)
+-AB*/CD-EF

Note: Postorder traversal, with the following implementation of visit :
- if operand PUSH
- if operator POP two operands, calculate, push result back

corresponds to arithmetic expression evaluation.

Example: Counting leaves in a tree

```
template <class T>
void CountLeafR(TreeNode<T> *t, int& count)
{
if (t!=NULL)
{//using postorder traversal
CountLeafR(t->Left(),count);
CountLeafR(t->Right(),count);
//visiting a node means incrementing if leaf
if (t->Left()==NULL&&t->Right()==NULL)
count++;
}
}

template <class T>
int CountLeaf ( TreeNode <T> *t)
{
  int countmytree=0;
  CountLeafR(mytree, countmytree);
  return Countmytree;
}
```

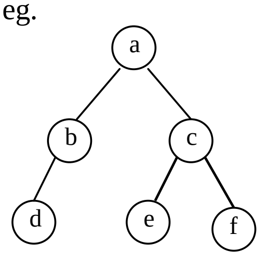
Example: computing depth of a tree

```
template <class T>
int Depth(TreeNode<T> *t)
int depthmytree ;
depthmytree=Depth(mytree) ;
{int depthleft, depthRight, depthval;
if (t==NULL)
 depthval=-1; // if empty, depth=-1
else
{ depthLeft=Depth(t->left());
   depthRight=Depth(t->Right());
  depthval=1+(depthleft>depthRight?depthLeft:depthRight));
}
 return depthval;
}
```

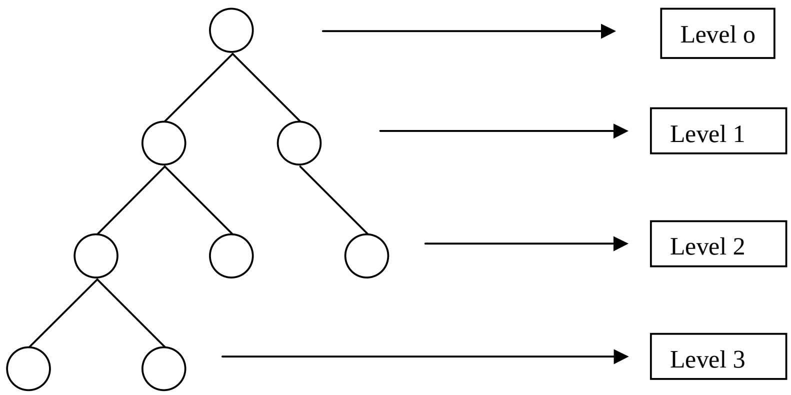
conditional expression syntax:
CONDITION? True-case-EXP:False-case-EXP

The preorder, postorder and inorder traversals are "depth-first"

A breath-first traversal algorithm
eg.



Traversal sequence a,b,c,d,e,f

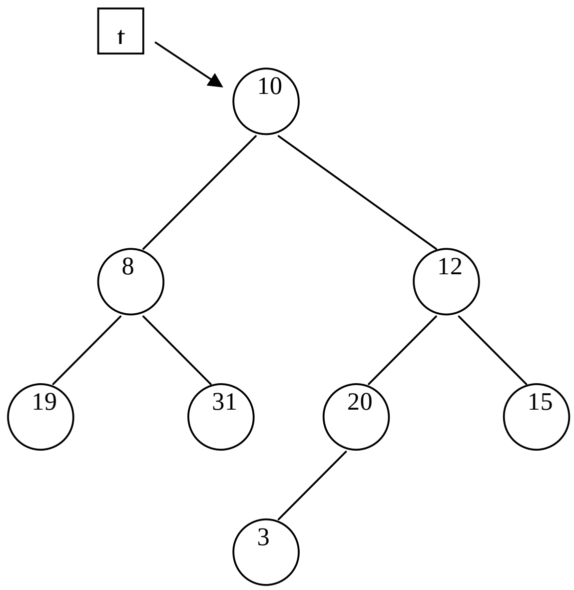


Find the level of the leaf node at minimum level.

```
template <class T>
int minLeafDepth( TreeNode <T> *t)
{
  if (t==NULL)
    return -1

  int levelleft, levelright, level;

  if (t->left ==NULL && t->right == NULL)
    return 0
  else {
    if (t->left != NULL)
      levelleft = minLeafDepth(t->left)
    else
      levelleft = maxint;
```

```
   if (t->right != NULL)
     levelright = minLeafDepth(t->right)
   else
     levelright = maxint;

   level = (levelleft < levelright ? levelleft : levelright) + 1;
   return level;
  }
}
```

A Binary Tree



Example: Find min key value stored in binary tree pointed by t. (assume tree is not empty)

```
template <class T>
int minval ( TreeNode <T> *t)
{
  int minval;
  if( t-> left ==NULL AND t->right == NULL)
    return t->data;
  else
  {
    if ( t->left != NULL)
      minleft = minval(t->left)
    else
      minleft = maxint;
    if ( t->right != NULL)
      minright = minval(t->right)
```

```
   else
     minright = maxint;

   return minval = (minval < t->data ? minval : t->data);
  }
}
```

Algorithm Level-Traverse

1. Insert root node in queue
2. while queue is not empty
    2.1. Remove front node from queue and visit it
    2.2. Insert Left child
    2.3. Insert right child

```
template <class T>
void LevelScan(TreeNode<T> *t, void visit(T& item))
{
queue<TreeNode<T>*> Q; // queue of pointers
TreeNode<T> *p;
// insert root
Q.Qinsert(t);
while(!Q.Empty())
{P=Q.Qdelete();
visit(P->data); //for ex. Cout P->data
if (p->Left()!NULL) Q.Qinsert(p->Left());
if (p->Right()!=NULL) Q.Qinser(p->Right());
}
}
```

M_WAY SEARCH TREE

Definition: An m_way search tree is a tree in which all nodes are of degree<=m. (It may be empty). A non empty m_way search tree has the following properties:

a) It has nodes of type:

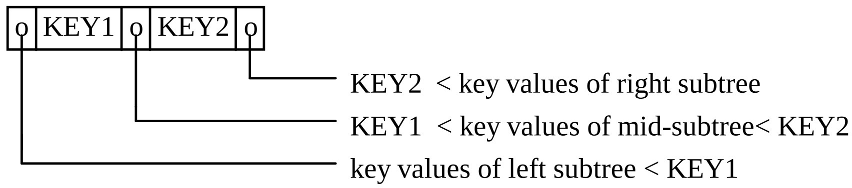
o	KEY1		KEY2	o	....	o	KEYm-1	o
↓		↓		↓		↓		↓
T1		T2		T3		Tm-1		Tm

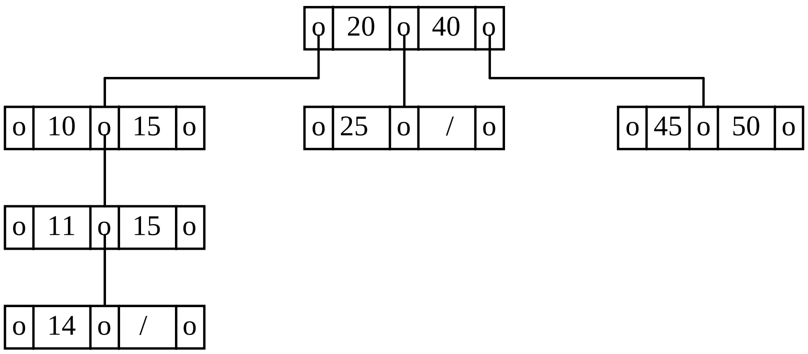
b) key1 < key2 <...< key(m-1)
  in other words, keyi<key(i+1), 1<=i<m-1
c) All Key values in subtree Ti are greater than Keyi and less than Keyi+1

Sometimes, we have an additional entry at the leftmost field of every node, indicating the number of nonempty key values in that node.

Example: 3_way search tree:

Nodes will be of type:

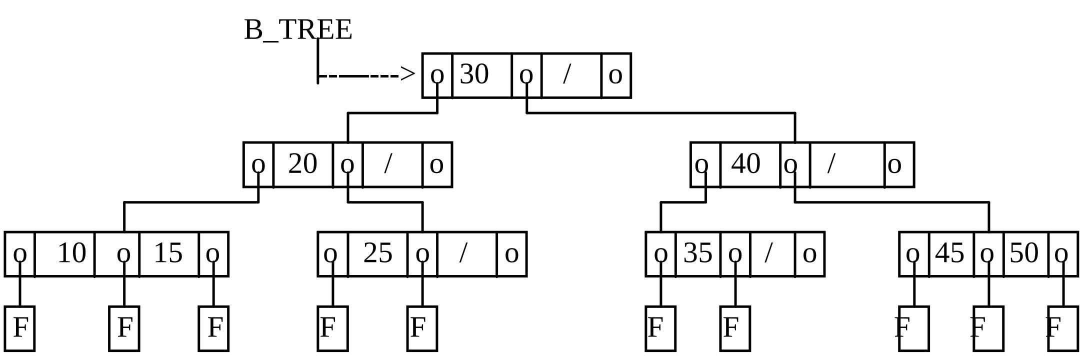




B_TREES

A B_tree of order m is an m_way search tree (possibly empty) satisfying the following properties (if it is not empty)
a) All nodes other than the root node and leaf nodes have at least m/2 children,
b) The tree is balanced. (If we modify all link fields of leaf nodes to point to special nodes called failure nodes are at the same level) (all the leaf nodes are at the same level)

Example: B_tree of order 3:



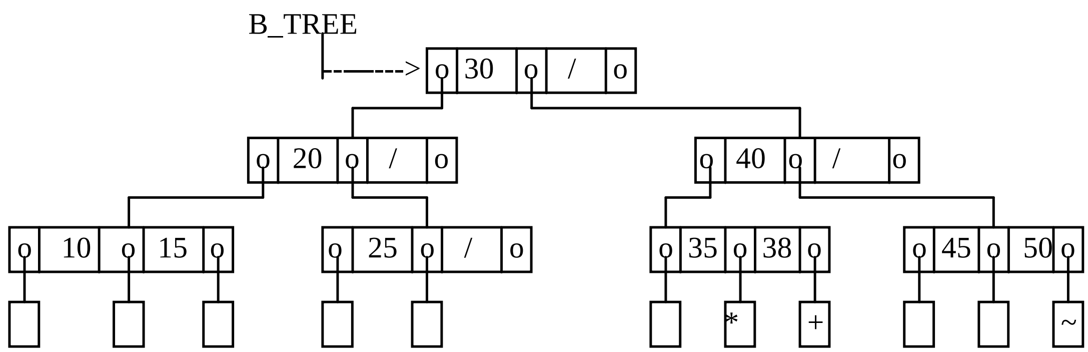
Inserting new key values to B_tree:

. We want to insert a new key value: x, into a B-tree
. The resulting tree must also be a B-tree. (It must be balanced.)
. We'll always insert at the leaf nodes.

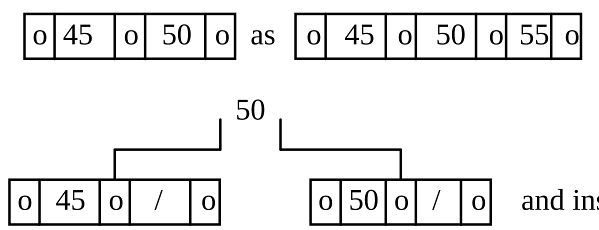
Example:

1) Insert 38 to the B_tree of the above example:

First of all, we do a search for 38 in the given b_tree. we hit the failure node marked with "*" . The parent of that failure node has only one key value so it has space for another one. Insert 38 there , add a new failure node, which is marked as "+" in the following figure, and return.



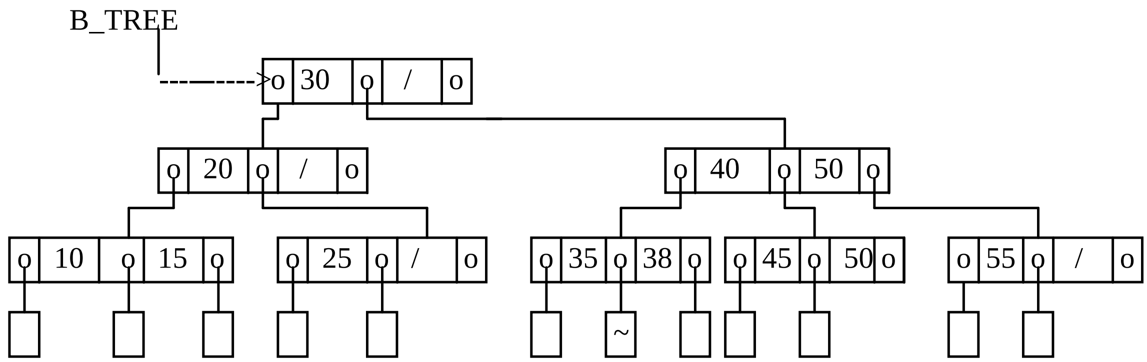
2) Now, insert 55 to this B_tree. We do the search and hit the failure node "~". However, it's parent node does not have any space for a key value. Now, assume we create a new node instead of



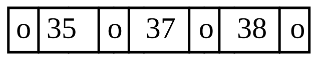
and split into two nodes

and insert 50 into parent node.

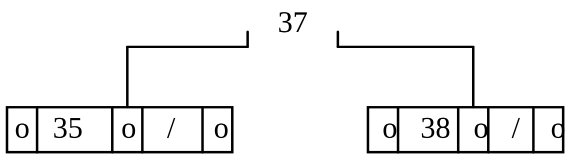
So we end up



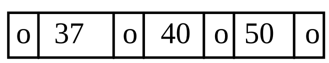
3) Now, let us insert 37 to this B_tree: We search for 37, and hit a failure node between 35 and 38. So,we have to create:



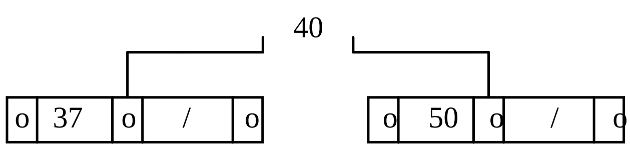
split it,



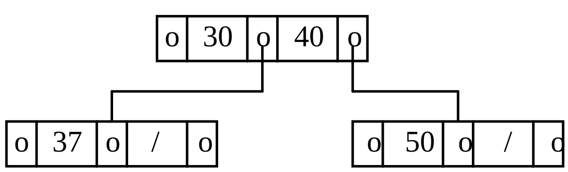
and insert 37 to its parent



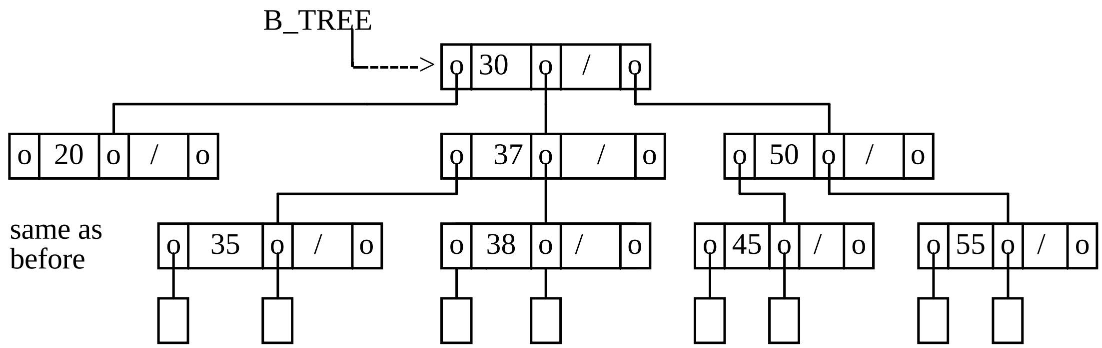
since there is no space for a new key , split it again



And this time insert 40 into its parent



So, we end up with



If we can not insert to the root node, we split and create a new root node. For example try to insert 34, 32, and 33. Thu, the height of the B_tree increases by 1 in such a case.

Deletion algorithm is much more complicated! It will not be considered here.

SORTING

We need to do sorting for the following reasons :
a) By keeping a data file sorted, we can do binary search on it.
b) Doing certain operations, like matching data in two different files, become much faster.

There are various methods for sorting, having different average and worst case behaviours.

	Best	Worst
Bubble sort	$\Omega(n^2)$	$O(n^2)$
Insertion sort	$\Omega(n^2)$	$O(n^2)$
Quick sort	$\Omega(n\log n)$	$O(n^2)$
Merge sort	$\Omega(n\log n)$	$O(n\log n)$

The average and worst case behaviours are given for a file having n elements (records).

1. Insertion Sort

Basic Idea:
Insert a record R into a sequence of ordered records: R1,R2,.., Ri with keys K1 <= K2 <= ... <= Ki , such that, the resulting sequence of size i+1 is also ordered with respect to key values.

```
Algorithm Insertion_Sort;       (* Assume Ro  has Ko = -maxint *)
void InsertionSort( Item &list[])
{ // Insertion_Sort
   Item r;
   int i,j;
   list[0].key = -maxint;
   for (j=2; j<=n; j++)
    {r=list[j];
      i=j-1;
      while ( r.key < list[i].key )
      {// move greater entries to the right
         list[i+1]:=list[i];
          i:=i-1;
       };
      list[i+1] = r  // insert into it's place
    }
```

We start with R0,R1 sequence, here R0 is artificial. Then we insert records R2,R3,..Rn into the sequence. Thus, the file with n records will be ordered making n-1 insertions.

Example: Let m be maxint

```
a)      j      R0  R1  R2  R3  R4  R5
        -----------------------
               -m   5   4   3   2   1
                      v
        2      -m   4   5   3   2   1
                          v
        3      -m   3   4   5   2   1
                              v
        4      -m   2   3   4   5   1

        5      -m   1   2   3   4   5
```

```
b)      j      R0  R1  R2  R3  R4  R5
        -----------------------
               -m  12   7   5  10   2
                      v
        2      -m   7  12   5  10   2
                          v
        3      -m   5   7  12  10   2
                                  v
        4      -m   5   7  10  12   2

        5      -m   2   5   7  10  12
```

2. Quick Sort: (C.A.R. Hoare, 1962)

Given a list of keys. Get the first key and find its exact place in the list. Carry the elements less than the first element to a sublist to the left and carry the elements greater than the first element to a sublist to the right.
Example:

```
 503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   3
  i   |   |   |   |   |   |   |   |   |   j   j   j   j   j   j
154<503
 154  |   |   |   |   |   |   |   |   |  503
154<->503
  i   i   i   |   |   |   |   |   |   |   j
503<512
         503  |   |   |   |   |   |   |  512
503<->512
          i   |   |   |   |   |   |   j   j
426<503
         426  |   |   |   |   |   |  503
426<->503
          i   i   i   |   |   |   |   j
503<908
                 503  |   |   |   |  908
503<->908
                  i   |   |   j   j   j
275<503
                 275  |   |  503
275<->503
                  i   i   i   j
503<897
                         503 897
503<->897
                          ij                                      i=j
stop
```

Now we have

```
        L1                                     L2
[154 087 426 061 275 170] 503 [897 653 908 512 509 612 677 765 703]
```

Aply quick sort to lists L1 and L2, recursively,

[061 087] 154 [170 275 426] 503 [....

And we get the sorted list following in this manner.

4. Radix Sort

Let's have the following 4-bit binary numbers. Assume there is no sign bit.

```
1010, 0101, 0011, 1011, 0110, 0111
(10)  (5)   (3)   (11)  (6)   (7)
```

1) First begin with the LSB (least significant bit). Make two groups,one with all numbers that end in a "0" and the other with all numbers that end in a "1".

```
  0         1
 ---       ---
1010      0101
0110      0011
          1011
          0111
```

2) Now, go to the next less SB and by examining the previous groups in order, form two new groups:

```
  0         1
 ---       ---
0101      1010
          0110
          0011
          1011
          0111
```

3) Repeat the operation for the third bit from the right:

```
  0         1
 ---       ---
1010      0101
0011      0110
1011      0111
```

4) Repeat it for the most significand bit:

```
  0         1                       result:  0011
 ---       ---                               0101
0011      1010                               0110
0101      1011                               0111
0110                                         1010
0111                                         1011
```

5. Merge Sort

In merge sort, two already sorted files are 'merged' to obtain a third file which is the sorted combination of the two sorted input files.

- We begin by assuming we have n sorted files with size=1
-Then,we merge these files of size=1 pairwise to obtain n/2 sorted files of size=2
-Then,we merge these n/2 files of size=2 pairwise to obtain n/4 sorted files of size=4,etc..
-Until we are left with one file with size=n.
Example :

```
   13       8       61     2      53     10      46     22
     \     /          \   /         \    /         \    /
     [8 13]           [2 61]       [10 53]        [22 46]
         \             /                \            /
         [2 8 13 61]                    [10 22 46 53]
                 \                           /
                 [2 8 10 13 22 46 53 61]
```

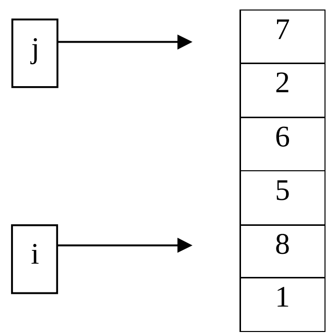
To merge two sorted files (x[1].. x[m]) and (y[1]..y[n]), to get a third file (z[1]..z [m+n]) with key1<key2<...<keyn , which will be the sorted combination of them, the following Pascal procedure can be used :

```
int MERGE(m,n:integer; int x[], int y[]; int &z[]);
int i,j,k,p
{ /* Merge */
  i=1; /* i is a pointer to x */
  j=m+1; /* j is a pointer to y */
  k:=1; /* k points to the next available location in z */
  while (i<=m)&& (j<=n)
   {
      if (x[i].key <= y[j].key)
      { /* take element from x */
        z[k]=x[i];
        i++;
      }
     else
       { /* take element from y */
         z[k]=y[j];
         j++;
       };
    k:=k+1 /* added one more element into z */
  }; /* while */

  if ( i>m )
  {
    for (p=j;nj<=n; j++)
          z[k++]= y[p]  /* remainig part of y into z */
  }
  else
```

```
{
        for (p=i; j<=m; j++)
              z[k++] = x[p] /* remaining part of x into z */
}

  return k-1;
}
```

BUBBLE SORT

```
void  Bubble_Sort(int  &A[], int n)
{
   int  i,j
   for (j= n-1; j>=1; j++)
     for (i=j;  i>=j; i--)
      if (A[i+1]<A[i]) { swap(A,i,i+1)}
}
```



QUESTIONS

---

1) a) Sort the following ternary (i.e. base 3) numbers using radix sort. Show all steps clearly.

1020, 1210, 2000, 0222, 0120, 1002, 1110, 1220.

b) What is the time complexity of radix sort on ternary numbers? Give it in terms of the number of elements to be sorted, the base (3 for ternary numbers), and the number of digits in elements to be sorted.

---

2) a) Sort the following integers using quick sort. Show all the steps clearly.

[12, 2, 16, 30, 8, 28, 5, 11, 19, 1, 46, 50]

b) What is the time complexity of the quick sort method?

---

3) a) How fast can we sort? Give the name and the time complexity of the best sorting method, among the sorting methods you have learned.

b) Write a Pascal procedure to search a given sorted one dimensional array of integers using the binary search method.

---

4) Write a pascal procedure which will merge three given files of integer key values of equal sizes. The input files are sorted in ascending order. The resulting file will again be in ascending order.

For example:

Given: [1,3,5,17] [2,7,9,11] [4,8,10,18]

The result will be: [1,2,3,4,5,7,8,9,10,11,17,18]

---

5) The following list of integers is to be sorted by quick sort:

[100, 59, 250, 361, 37, 1, 568, 21, 3].

Find the numbers to be placed into sublist L1 and L2 in the correct order, after the first step of the quick sort procedure is applied. (L1) 100 (L2)

---

7) Consider the following sorting methods:

insertion sort, quick sort, heap sort, radix sort and merge sort.

a) Suppose you are going to sort a list of integers L[n], consisting of sublists L133L2, where L1 has p elements, and L2 has q elements such that p+q=n, and p>>q. L1 is already sorted, but L2 is not sorted. Which of the above sorting methods is suitable for sorting list L? Why?

b) A sorting method is stable if it preserves the original order of records with equal key values. which of the above sorting methods are stable.

----

8) Consider the following algorithm to sort a sequence of elements a1, a2, a3, ..., an stored in array A.

```
Procedure Bubble_Sort(var A: array [1..n] of integer);
var i,j: integer;
begin (* bubble sort *)
   for j := n-1 downto 1 do
     for i := 1  downto j do
      if (A[i+1]<A[i]) then swap(A,i,i+1)
end; (* bubble sort *)
```

Where swap(A,k,l) is a procedure which interchanges the k'th and the l'th elements of array A.

a) Assume initially A=[7,2,9,5,3]. Write the contents of A for each different value of j, just after the inner for loop is executed.

b) Determine (in terms of n), how many times the swap procedure will be called in the worst case.
c) Assuming n=5, give an array A which achieves the worst case discussed in part b.

HASH CODING AND HASH TABLES

Hashing is a method of storing records according to their key values. It provides access to stored records in constant time, O(1), so it is comparable to B-trees in searching speed.

Therefore, hash tables are used for:

a) Storing a file record by record.
b) Searching for records with certain key values.

In hash tables, the main idea is to distribute the records uniquely on a table, according to their key values. We take the key and we use a function to map the key into one location of the array: f(key)=h, where h is the hash address of that record in the hash table.

If the size of the table is n, say array [1..n], we have to find a function which will give numbers between 1 and n only.

Each entry of the table is called a bucket. In general, one bucket may contain more than one (say r) records. In our discussions we shall assume r=1 and each bucket holds exactly one record.

Definitions:

key density:

k = n * r (hash table size)) / N (no of distinct possible key values)

loading factor :

LF = i (number of items in the table) / r * n (hash table size)

Two key values are synonyms with respect to f, if f(key1)=f(key2).

Synonyms are entered into the same bucket if r>1 and there is space in that bucket.

When a key is mapped by f into a full bucket this is an overflow.

When two nonidentical keys are mapped into the same bucket, this is a collision.

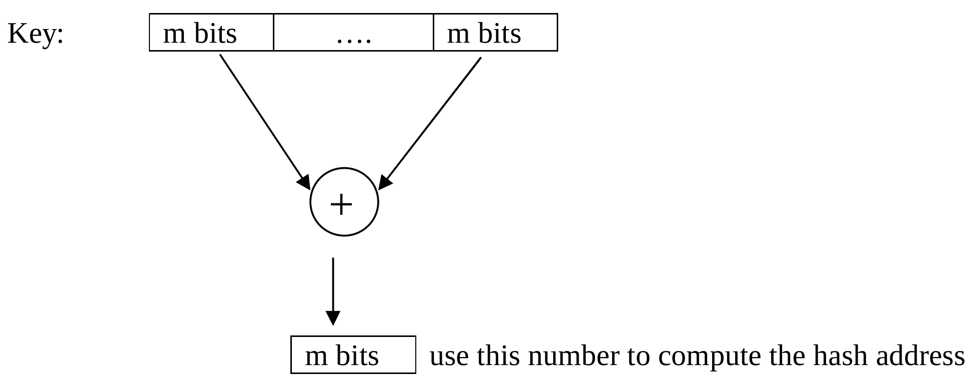
The hash function f,

a) Must be easy to compute,
b) Must be a uniform hash function.
(a random key value should have an equal chance of hashing into any of the n buckets.)
b) Should minimize the number of collisions.

Some hash functions used in practical applications :

1) f(key)=key mod n can be a hash function,

However n should never be a power of 2, n should be a prime a number.
2) Ex-or'ing the first and the last m bits of the key:



Notice that the hash table will now have a size n, which is a power of 2.

3) Mid-squaring:

a) take the square of the key.
b) then use m bits from the middle of the square to compute the hash address.

4) Folding:

The key is partitioned into several parts. All exept the last part have the same length. These parts are added together to obtain the hash address for the key. There are two ways of doing this addition.

a) Add the parts directly
b) Fold at the boundaries.

Example:

key = 12320324111220,   part length=3,

123	203	241	112	20
P1	P2	P3	P4	P5

a)	b)
123	123
203	302
241	241
112	211
+ 20	+ 20
699	897

Handling Collisions - Overflows :

Consider r=1, so there is one slot per bucket. All slots must be initialized to 'empty' (for instance, zero or minus one may denote empty).

1) Linear probing:

0		f(key1)=2
1		f(key2)=2, but location 2 is full.
2	key1	So, go to the next empty location and store key2 there
3	key2	But now if f(key3)=2, another collision!.

- When we reach the end of the table, we go back to location 0.
- Finding the first empty location will sometimes take a lot of time.
- Also, in searching for a specific key value, we have to continue the search until we find an empty location, if that key value is not found at the calculated hash address.

2) Random probing

When there is a collision, we start a (pseudo) random number generator.
For example,

f(key1)=3
f(key2)=3 $\rightarrow$ collision

Then, start the pseudo random number generator and get a number, say 7. Add 3+7=10 and store key2 at location 10.

The pseudo-random number i is generated by using the hash address that causes the collision. It should generate numbers between 1 and n and it should not repeat a number before all the numbers between 1 and n are generated exactly once.

In searching, given the same hash address, for example 3, it will give us the same number 7, so key2 shall be found at location 10.

We carry out the search until:

a) We find the key in the table,
b) Or, until we find an empty bucket, (unsuccessful termination)
c)Or, until we search the table for one sequence and the random number repeats. (unsuccessful termination, table is full)

3. Chaining

We modify entries of the hash table to hold a key part (and the record) and a link part. When there is a collision, we put the second key to any empty place and set the link part of the first key to point to the second one. Additional storage is needed for link fields.

0		
1		
2	key1	6
3		
4		
5		
6	key2	null
7		

f(key1)=3
f(key2)=3 $\rightarrow$collision, Put key2 to bucket 6
But now, what happens if f(key3)=6 ?
Take key2 out, put key3 to bucket 6,.and then put key2 to another available bucket and change link of key1.

4) Chaining with overflow

In this method, we use extra space for colliding items.

HT

0		
1		
2		
3	key 1	
4		
5		
6	key 3	
7		

Overflow area

0	key2	2
1	key4	null
2	key5	null
3		
4		
5		
6		
7		

f(key1)=3 goes into bucket 3
f(key2)=3 collision, goes into the overflow area
f(key3)=6 goes into bucket 6
f(key4)=6 collision, goes into the overflow area
f(key5)=3 collision, goes to over flow area with chain

6) Rehashing :

Use a series of hash functions. If there is a collision, take the second hash function and hash again, etc... The probability that two key values will map to the same address with two different hash functions is very low.

Average number of probes (AVP) calculation :

Calculate the probability of collisions, then the expected number of collisions, then average. (See Horowitz and Sahni)

1. Linear probing :

AVP = (1-LF/2) / (1-LF) where LF is the loading factor.

2. Random probing :

AVP = (1/LF)*ln(1-LF)

3. Chaining with overflow

AVP = 1 + (LF/2)

LF	LINEAR P	RANDOM P	CHAINING W.O.
0.1	1.06	1.05	1.05
0.5	1.50	1.39	1.25
0.9	5.50	2.56	1.45

Deleting key values from HT's :

0	
1	
2	key1
3	key2

Consider linear probing,
f(key1)=2
f(key2)=2

To delete key1, we have to put a special sign into location 2 because there might have been and we can break the chain if we set thatbucket to empty. However then we shall be wasting some empty locations, LF is increased and AVP is increased. We can not increase the hash table size, since the hash function will generate values between 1 and n (or, 0 and n-1). Using an overflow area is one solution

.

QUESTIONS

----

1) We would like to keep a table of last names of all the students of METU. The table shall be formed of records of the type: <last_name, occurance>
We would like to use a hash table for this purpose for fast insertion and search. Keys for records are the last-name fields shall be held as hexadecimal numbers, obtained by converting each letter to its two-digit hexadecimal code.
For example for the last name 'AKAN', assuming the hex. code of 'A' is '60', the hex. code for 'K' is '6C' and hex. code of 'N' is '6F', its representation will be the hex. number '606C606F'. Assume bucket size=1.

a) Given that the number of students is currently 18000, but may increase in the future, up to 25000, choose an appropriate hash table size.

b) Make a choice from the following two hashing methods for this system and explain your reasoning.

i) hash function: key mod (table size)
method :linear probing

ii) hash function :mid-squaring
method :chaining with overflow area.

c) What should be done in case a new student is registered?

d) What should be done in case a student graduates or leaves the university?

Illustrate your answers for parts a,c, and d with tables.

----

2) a) What are the basic properties that a hash function should satisfy?

b) Give an algorithm to delete key values from a hash table of size n, assuming bucket size is one, and linear probing is used to handle collisions. The hash function f is available as a library function, and it returns the hash address for any given key value.

c) Repeat part b for random probing.

d) Compare the algorithms you gave in parts b and c, and discuss their relative efficiency.

----

3) Consider a hash table which has m entries and an overflow area of size s. Chaining with overflow area method is used for handling overflows. Bucket size is k, where k>1. The hash function is, f(key)= (key mod m)

a) Write a SEARCH_AND_INSERT(key,found, index) procedure in Pascal which will insert a given key value into the hash table, if it is not already there.

b) Write a DELETE(key) procedure which will delete a given key value from the hash table, if it is there. You may make use of procedures/functions written for part 'a'.

----

4) We have a hash table containing records with positive integer key values in the range [1..1000].

The random probing method is used, hash function is : key mod n, bucket size is one.

Give an algorithm to list all key values present in this hash table, in increasing order. Find the time complexity of your algorithm.

----

5) Consider a hash table which has m entries. Linear probing method is used for handling overflows. Bucket size is one. The hash function is :
f(key)=(key mod m)

a) Write a SEARCH(key,found,index) procedure in Pascal which will search a given key value in the hash table. If the key value is found in the table then, it will return a true value in the parameter '**found**', and the index at which the key value was found in the parameter '**index**'. Otherwise, found will be false and index will contain the index of the last location checked, before deciding that the key is not in the table.

b) Write an INSERT(key) procedure which will insert a given key value into the hash table, if it is not already there. you may call the SEARCH procedure you wrote in part "a" in INSERT.

c) Write a DELETE(key) procedure which will delete a given key value from the hash table, if it is there. Again, you may call the SEARCH procedure in DELETE.
