

**SYMBIOSIS INSTITUTE**

**OF TECHNOLOGY, PUNE**

**BANKER’S ALGORITHM AND PAGE REPLACEMENT ALGORITHMS**

***SUBMITTED ON: 22nd APRIL, 2015***

**PROJECT TEAM:**

**NisargDhuvad PRN:13070121124**

**PratikPatil PRN:13070121129**

**Sanchita Seth PRN:13070121138**

**Shiwani Singh PRN:13070121148**

# Data Structure used : Arrays

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.  
  
That means that, for example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.  
  
For example, an array containing 5 integer values of type int called foo could be represented as:  
  
http://www.cplusplus.com/doc/tutorial/arrays/arrays1.png   
where each blank panel represents an element of the array. In this case, these are values of type int. These elements are numbered from 0 to 4, being 0 the first and 4 the last; In C++, the first element in an array is always numbered with a zero (not a one), no matter its length.  
  
Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:  
  
type name [elements];  
  
where type is a valid type (such as int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies the length of the array in terms of the number of elements.  
  
Therefore, the foo array, with five elements of type int, can be declared as:

|  |  |  |
| --- | --- | --- |
|  | int foo [5]; |  |

## NOTE: The elements field within square brackets [], representing the number of elements in the array, must be a*constant expression*, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs. Two-Dimensional Arrays:

**2 D Arrays : (Used in LFU and Banker’s)**

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows:

type arrayName [ x ][ y ];

Where **type** can be any valid C++ data type and **arrayName** will be a valid C++ identifier.

A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below:

Thus, every element in array a is identified by an element name of the form **a[ i ][ j ]**, where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

## Initializing Two-Dimensional Arrays:

## C:\Users\user\Downloads\two_dimensional_arrays.jpg

Multidimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

int a[3][4] = {

{0, 1, 2, 3} , /\* initializers for row indexed by 0 \*/

{4, 5, 6, 7} , /\* initializers for row indexed by 1 \*/

{8, 9, 10, 11} /\* initializers for row indexed by 2 \*/

};

The nested braces, which indicate the intended row, are optional.

int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};

**Accessing Two-Dimensional Array Elements:**

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

int val = a[2][3];

The above statement will take 4th element from the 3rd row of the array.You can verify it in the above digram.

#include <iostream.h>

int main ()

{

int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

for ( int i = 0; i < 5; i++ )

for ( int j = 0; j < 2; j++ )

{

cout << "a[" << i << "][" << j << "]: ";

cout << a[i][j]<< endl;

}

return 0;

}

**BANKER’S ALGORITHM**

The **Banker's algorithm** is a resource allocation and deadlock avoidance algorithm developed by EdsgerDijkstra that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.



**CODE:-**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

intavilable[20];

int max[20][20];

int allocation[20][20];

int need[20][20];

voidbankersalgo()

{

intn,m,i,j,cntr=0,allow=0,no,req1,req2,req3;

int flag[20]={0},f=0;

printf("\n\tEnter number of processes : ");

scanf("%d",&n);

printf("\n\tEnter number of Resources : ");

scanf("%d",&m);

printf("\nEnter allocation matrix : \n\t");

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

{

printf("\n\t");

for(j=0;j<m;j++)

{

scanf("%d",&allocation[i][j]);

}

}

printf("\nEnter MAX matrix : \n\t");

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

{

printf("\n\t");

for(j=0;j<m;j++)

{

scanf("%d",&max[i][j]);

}

}

printf("\n\tEnter Available : ");

for(i=0;i<m;i++)

{

printf("\n\tEnter available value of resource %d : ",i);

scanf("%d",&avilable[i]);

}

printf("\n\t Calculating need....");

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

{

for(j=0;j<m;j++)

{

need[i][j]=max[i][j]-allocation[i][j];

}

}

printf("\n\tNeed calculation completed...");

cntr=0;

while(1)

{

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

{

allow=0;

for(j=0;j<m;j++)

{

if(need[i][j]<= avilable[j]&&flag[i]==0)

allow=1;

else

{

break;

}

}

if(allow==1)

{

printf("\n\t\*\*\*process P%d is safe\*\*\*",i);

avilable[0]+=allocation[i][0];

avilable[1]+=allocation[i][1];

avilable[2]+=allocation[i][2];

flag[i]=1;

cntr++;

}

}

if(cntr==n)

break;

}

//Resource Request Allocation

while(1)

{

printf("\nEnter Requested process : ");

scanf("%d",&no);

printf("\nEnter Requested resources : ");

scanf("%d%d%d",&req1,&req2,&req3);

if(req1<=need[no][0]&&req2<need[no][1]&&req3<need[no][2])

if(req1<avilable[0]&&req2<avilable[1]&&req3<avilable[2])

{

printf("\n\t!!!!Granted requested!!!!!");

avilable[0]=avilable[0]-req1;

avilable[1]=avilable[1]-req2;

avilable[2]=avilable[2]-req3;

allocation[no][0]=allocation[no][0]+req1;

allocation[no][1]=allocation[no][1]+req2;

allocation[no][2]=allocation[no][2]+req3;

need[no][0]=need[no][0]-req1;

need[no][1]=need[no][1]-req2;

need[no][2]=need[no][2]-req3;

}

else

printf("\n!!!Process p %d must be waited!!!",no);

else

printf("Error...exceding request allocation \*\*\*");

printf("\n\tDou want to continue(YES=1/No=0) : ");

scanf("%d",&f);

if(f==0)

break;

}

}

**PAGE REPLACEMENT ALGORITHMS**

In a computer operating system that uses paging for virtual memory management, **page replacement algorithms** decide which memory pages to page out (swap out, write to disk) when a page of memory needs to be allocated. Paging happens when a page fault occurs and a free page cannot be used to satisfy the allocation, either because there are none, or because the number of free pages is lower than some threshold.

Following are the various page replacement algorithm:-

1. FIFO(First In First Out)

In this algorithm the operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the oldest arrival in front. When a page needs to be replaced, the page at the front of the queue (the oldest page) is selected.

Code:-

#include<iostream>

#include<stdio.h>

#include<stdlib.h>

#include<ctype.h>

using namespace std;

intfoundlru(intx,int \*l,int max)

{

for(inti=0;i<max;i++)

if(l[i]==x){return(i);}

return(-1);

}

voidfifoalgorithm()

{

cout<<"\n\nEnter the maximum number of frames in the main memory:\t";

int max;

cin>>max;

int \*l=new int[max];

for(inti=0;i<max;i++)l[i]=-1;

inta,x;

int k=0,c=0,res;

cout<<"\n\nEnter the sequence of page requests(enter -1 to stop):\t";

while(1)

{

cin>>x;

if(x==-1) {cout<<"\n\n";break;}

else{

if(k<max)

{

if((res=foundlru(x,l,max))!=-1) {cout<<"\n\npage "<<x<<" already exists in frame "<<res<<" in MM";

cout<<"\n\nNext page:\t";}

else

{

cout<<"\n\npage "<<x<<" has been allocated a frame "<<k<<" in MM.";

l[k++]=x;

cout<<"\n\nNext page:\t";

}

}

else

{

if((res=foundlru(x,l,max))!=-1) {cout<<"\n\npage "<<x<<" already exists in frame "<<res<<" in MM";

cout<<"\n\nNext page:\t";}

else{

cout<<"\n\npage fault has occured";

cout<<"\n\npage "<<x<<" has been allocated frame "<<c<<" in MM by replacing page "<<l[c];

l[c]=x;

c=(c+1)%max;

cout<<"\n\nNext page:\t";

}

}

}

}

delete[] l;

}

**2.LFU(Least Frequently used)**

**Least Frequently Used** (**LFU**) is a type of cache algorithm used to manage memory within a computer. The standard characteristics of this method involve the system keeping track of the number of times a block is referenced in memory. When the cache is full and requires more room the system will purge the item with the lowest reference frequency.

CODE:-

#include<stdio.h>

voidLFUalgo()

{

intf,p;

int pages[50],frame[10],hit=0,count[50],time[50];

inti,j,page,flag,least,minTime,temp;

printf("Enter no of frames : ");

scanf("%d",&f);

printf("Enter no of pages : ");

scanf("%d",&p);

for(i=0;i<f;i++)

{

frame[i]=-1;

}

for(i=0;i<50;i++)

{

count[i]=0;

}

printf("Enter page no : \n");

for(i=0;i<p;i++)

{

scanf("%d",&pages[i]);

}

printf("\n");

for(i=0;i<p;i++)

{

count[pages[i]]++;

time[pages[i]]=i;

flag=1;

least=frame[0];

for(j=0;j<f;j++)

{

if(frame[j]==-1 || frame[j]==pages[i])

{

if(frame[j]!=-1)

{

hit++;

}

flag=0;

frame[j]=pages[i];

break;

}

if(count[least]>count[frame[j]])

{

least=frame[j];

}

}

if(flag)

{

minTime=50;

for(j=0;j<f;j++)

{

if(count[frame[j]]==count[least] && time[frame[j]]<minTime)

{

temp=j;

minTime=time[frame[j]];

}

}

count[frame[temp]]=0;

frame[temp]=pages[i];

}

for(j=0;j<f;j++)

{

printf("%d ",frame[j]);

}

printf("\n");

}

printf("Page hit = %d",hit);

}

**3.LRU(Least Recently Used)**

In this algorithm it replaces the page that has not been

referenced for the longest time

CODE:-

#include<iostream>

#include<stdio.h>

#include<stdlib.h>

#include<ctype.h>

#include<conio.h>

using namespace std;

intmindis(int \*dis,int max)

{

int m=0;

for(inti=0;i<max;i++)

{

if(dis[i]<dis[m]) m=i;

}

return(m);

}

int found(intx,int \*l,int max)

{

for(inti=0;i<max;i++)

if(l[i]==x){return(i);}

return(-1);

}

voidLRUAlgo()

{

intmax,x,k=0,j=0,res;

cout<<"\n\nEnter the maximum number of frames in the main memory:\t";

cin>>max;

int \*l=new int[max];

int \*a=new int[1000000];

for(inti=0;i<max;i++)l[i]=-1;

cout<<"\n\nEnter the sequence of page requests(enter -1 to stop):\t";

int \*dis=new int[max];

int \*flag=new int[max];

for(inti=0;i<max;i++) flag[i]=0;

while(1)

{

cin>>x;

if(x==-1) {cout<<"\n\n";break;}

else{

if(j<max)

{

if((res=found(x,l,max))!=-1) {cout<<"\n\npage "<<x<<" already exists in frame "<<res<<" in MM";

cout<<"\n\nNext page:\t";a[k++]=x;}

else

{

cout<<"\n\npage "<<x<<" has been allocated frame "<<j<<" in MM.";

l[j++]=a[k++]=x;

cout<<"\n\nNext page:\t";

}

}

else

{

if((res=found(x,l,max))!=-1) {cout<<"\n\npage "<<x<<" already exists in frame "<<res<<" in MM";

cout<<"\n\nNext page:\t";a[k++]=x;}

else{

cout<<"\n\npage fault has occured";

for(int z=0;z<max;z++) flag[z]=0;

for(int p=0;p<max;p++)

for(int q=k-1;q>=0;q--)

{

if(l[p]==a[q] && flag[p]==0) {dis[p]=q;flag[p]=1;}

else if(flag[p]!=1) dis[p]=-1000000;

}

intpos=mindis(dis,max);

cout<<"\n\npage "<<x<<" been allocated frame "<<pos<<" in MM by replacing page "<<l[pos];

l[pos]=x;

a[k++]=x;

cout<<"\n\nNext page:\t";

}

}

}

}

getch();

}

**Referances :**

[**http://en.wikipedia.org/wiki/Page\_replacement\_algorithm#References**](http://en.wikipedia.org/wiki/Page_replacement_algorithm#References)

[**http://en.wikipedia.org/wiki/Banker%27s\_algorithm**](http://en.wikipedia.org/wiki/Banker%27s_algorithm)

**Conclusion :**

In this project, we implemented important concepts of Operating Systems i.e. , The Banker’s algorithm and Page replacement algorithms (LFU, LRU, FIFO). While implementing these concepts, we realized how basic algorithms control the operation of such complicated systems like the Operating Systems and how important they can be in circumstances where the system could simply go into a deadlock state, or gets confused which page to replace when it has no free frames

The coded versions of these algorithms can help resolve such dicey issues in a very short time. This is important in situations which require operations to be performed in nano seconds.

We thank Shruti Mam, without whose detailed explanation these concepts couldn’t have been implemented.