Sa se implementeze o clasa ce va implementa o strategie de alocare de timp procesor pentru procesele aflate in memorie la un moment dat, asemanator functionarii unui planificator (scheduler) traditional din cadrul unui sistem de operare multitasking.

```
================ Task.h ============
#ifndef _TASK_
#define _TASK_

#include <iostream.h>
#include <stdlib.h>
#include <dos.h>

class Process{
            int id;
            long priority;
      public:
            Process(int, long);
            long getPriority(void);
            void setPriority(long p);

            virtual void execute();
};

#endif

=============== Task.cpp ==========
#include "task.h"

#define MAX_RUN 2000

Process::Process(int _id, long _priority) {
  id = _id;
  priority = _priority;
}

long Process::getPriority(void) {
  return priority;
}

void Process::setPriority(long p) {
  priority = p;
}

void Process::execute(void) {
  int time;

  time = 1 + random(MAX_RUN);
  cout << "Procesul " << id <<  " ruleaza " << time << " milisecunde.\n";
  delay(time);
}
```

=============== PQueue.h ==========

```
#ifndef _PQUEUE_
#define _PQUEUE_

#include <mem.h>
#include "task.h"

#define MAX_SIZE 100
```

```
class PriorityQueue{
            Process* elements[MAX_SIZE + 1];
            int last;
      public:
            PriorityQueue();

            Process* deleteMin(void);
            void insertElement(Process*);
            void initQueue(void);
};

#endif
```

========== PQueue.cpp ==========

```
#include "pqueue.h"

PriorityQueue::PriorityQueue() {
  initQueue();
}

Process* PriorityQueue::deleteMin(void) {
  int i, j;
  Process *tmp;
  Process *minimum;

  if (!last)
    cout << "Coada vida! \n";
  else {
    minimum = elements[1];
    elements[1] = elements[last];
    last --;
    i = 1;
    while (i <= last/2) {
      if (elements[2*i]->getPriority() < elements[2*i+1]->getPriority() || 2*i
== last)
        j = 2*i;
      else
        j = 2*i + 1;
      if (elements[i]->getPriority() > elements[j]->getPriority()) {
        tmp = elements[i];
        elements[i] = elements[j];
        elements[j] = tmp;
        i = j;
      }
      else
        return minimum;
    }
    return minimum;
  }
  return NULL;
}

void PriorityQueue::initQueue(void) {
  last = 0;
}

void PriorityQueue::insertElement(Process* t) {
  int i;
  Process* tmp;

  if (last >= MAX_SIZE)
```

```
    cout << "Coada plina!\n";
  else {
    last++;
    elements[last] = t;
    i = last;
    while (i > 1 && elements[i]->getPriority() < elements[i/2]->getPriority()) {
      tmp = elements[i];
      elements[i] = elements[i/2];
      elements[i/2] = tmp;
      i /= 2;
    }
  }
}
```

========== Manager.h ==========

```
#ifndef _MANAGER_
#define _MANAGER_

#include "pqueue.h"

class Manager {
int n;

            long* currenttime;
            PriorityQueue waiting;
            void selectProcess(void);
      public:
            Manager(int);

            void run();
};

#endif
```

========== Manager.cpp ==========

```
#include "manager.h"
#include <conio.h>

Manager::Manager(int _n) {
  n = _n;
  currenttime = (long*)MK_FP(0x0, 0x46c);
  for (int i = 0; i < n; i++) {
    Process* process = new Process( i + 1, -*currenttime);
    waiting.insertElement(process);
  }
}

void Manager::selectProcess(void) {
  long begintime,endtime;
  Process* process;

  process = waiting.deleteMin();
  begintime = *currenttime;
  process -> execute();
  endtime = *currenttime;
  process-> setPriority(process -> getPriority() + 100 * (endtime - begintime));
  waiting.insertElement(process);
}
```

```
void Manager::run(void) {
  while (!kbhit())
    selectProcess();
  cout << "Stop!\n";
}
```

========== Simula.cpp ==========

```
#include "manager.h"

void main(void) {
  int m;

  cout << "Numarul de procese: ";
  cin >> m;

  Manager manager(m);

  manager.run();
}
```

Sa se implementeze o clasa care va modela comportamentul unui arbore binar de sortare. Operatiile suportate sunt:

- inserarea unui nod in arbore
- cautarea dupa o anumita cheie
- stergerea unui nod corespunzator unei anumite chei daca aceasta exista in arbore
- parcurgerea in inordine a arborelui.

========== BinSTree.h ==========

```
#ifndef _BINSTREE
#define _BINSTREE

#include <stdlib.h>
#include <string.h>
#include <iostream.h>

class Nod {
   char* m_key;
   void* m_reference;
   Nod* left;
   Nod* right;

 public:
   Nod(char* key,void* reference) {
     m_key = new char [sizeof(char)*(strlen(key)+1)];
     strcpy(m_key,key);
     m_reference = reference;

     left = NULL;
     right = NULL;
   }

  friend class BinSTree;
};

class BinSTree {
    Nod* root;
    Nod* nodInsert(Nod* p,char* key, void* obj);
```

```
    Nod* LeftmostNod(Nod* parent, Nod* curent);
    Nod* deleteNod(Nod* p, char* key);
    void inordine(Nod* p, int level);

  public:
    BinSTree();

    void insert(char* key, void* obj);
    void* search(char* key);
    void remove(char* key);
    void inordine();
};

#endif
```

================== BinSTree.cpp ==========

```
#include "binstree.h"
#include "student.h"

/* ToDo:
   Sa se implementeze:
     - constructor de copiere
     - destructorul
     - functiile de vizitare in preordine si postordine
     - operatorul "="
     - operatorul ">>"  cu rol de stergere nod
     - "[]" intoarce al i-lea nod la parcurgerea in inordine
*/

/* Constructorul clasei*/
BinSTree::BinSTree() {
  root = NULL;
}

/* Insereaza un element pe pozitia corespunzatoare cheii.
   Parametrii:
     key - cheia de inserare (sir de caractere)
     obj - pointer catre un obiect
*/
void BinSTree::insert(char* key,void* obj) {
  root = nodInsert(root, key, obj);
}

/* Insereaza un element intr-un arbore de radacina p. Functie recursiva.
   Parametrii:
     p   - radacina arborelui
     key - cheia de inserare (sir de caractere)
     obj - pointer catre obiectul ce contine informatia primara
*/
Nod* BinSTree::nodInsert(Nod* p, char* key, void* obj) {
  if (p == NULL)
    return new Nod(key, obj);
  else {
    int c = strcmp(key, p->m_key);
    if (c < 0)
      p->left = nodInsert(p->left, key, obj);
    else
      if (c > 0)
        p->right = nodInsert(p->right, key, obj);
  }
  return p;
```

```
}

/* Cauta informatia atasata unei chei. Cautarea este nerecursiva.
   Parametrii:
     key - cheia
*/
void* BinSTree::search(char* key) {
  Nod* p = root;
  int c;

  while (p !=  NULL) {
    c = strcmp(key,p->m_key);
    if (c < 0)
      p = p->left;
    else
      if (c > 0)
        p = p->right;
      else
        break;
  }
  if (p != NULL)
    return p->m_reference;

  return NULL;
}

/* Intoarce cel mai din stinga nod al descendentului drept al unui nod.
   Parametrii:
     parent - nodul parinte
     curent - descendentul drept al nodului parent
*/
Nod* BinSTree::LeftmostNod(Nod* parent, Nod* curent) {
  while (curent->left != NULL) {
    parent = curent;
    curent = curent->left;
  }
  if (parent->right == curent)
    parent->right = curent->right;
  else
    parent->left = curent->right;

  return curent;
}

/* Sterge nodul de cheie key din arborele de radacina p.
   Parametrii:
     p   - radacina arborelui
     key - cheia
*/
Nod* BinSTree::deleteNod(Nod* p, char* key) {
  Nod* tmp;
  int ind;

  if (p != NULL) {
    ind = strcmp(p->m_key, key);
    if (ind < 0)
      p->right = deleteNod(p->right, key);
    else
      if (ind > 0)
        p->left = deleteNod(p->left, key);
      else
```

```
        if (p->left == NULL || p->right == NULL) {
          if (p->left != NULL)
            tmp = p->left;
          else
            tmp = p->right;
          p = tmp;
        }
        else {
          tmp = p->right;
          tmp = LeftmostNod(p, tmp);
          tmp->left = p->left;
          tmp->right = p->right;
          p = tmp;
        }
  }

  return p;
}

/* Elimina un nod de cheie specificata dintr-un arbore.
   Parametrii:
     key - cheia
*/
void BinSTree::remove(char* key) {
  root = deleteNod(root, key);
}

/*  Viziteaza in inordine si afiseaza informatia asociata nodurilor arborelui.
    Aici este singurul loc din program in care arborele trebuie "sa stie" de
clasa Student.
    Parametrii:
       p  - nodul curent
      level - "adincimea" nodului curent
*/
void BinSTree::inordine(Nod* p, int level) {
  if (p != NULL) {
    inordine(p->left, level+1);
    for (int i = 0; i < level; i++)
      cout << "  ";
    ((Student*)p->m_reference)->afis();
    inordine(p->right, level+1);
  }
}

/* Parcurge in inordine arborele.*/
void BinSTree::inordine() {
   cout<<"----------------------------\n";
   inordine(root, 1);
}
```

========== Student.h ==========

```
#ifndef _STUDENT
#define _STUDENT

#include <stdlib.h>
#include <iostream.h>
#include <string.h>

class Student {
    int   m_matricol;
    char *m_nume;
```

```
    char *m_adresa;
    void makefree();

  public:
    Student();
    Student(char *nume, char* adresa, int matricol);
    Student(Student& s);
    ~Student();

    void afis();
    char* getName();
};

#endif
```

================== Student.cpp ==============

```
#include "student.h"

/* TODO:
    Sa se implementeze:
      - functiile de access la variabilele private
      - operatorul "="
      - operatorul "<<" pentru afisare
*/

/* Constructor fara parametrii pentru clasa student*/
Student::Student() {
  m_nume = "John Doe";
  m_adresa = "Unknown";
  m_matricol = 1;
}

/*Constructor obisnuit:
  parametrii:
    nume - numele studentului
    adresa - adresa (strada,nr,oras etc)
    matricol - numarul matricol
*/
Student::Student(char* nume, char* adresa, int matricol) {
  if (nume != NULL) {
    m_nume = new char [sizeof(char)*(strlen(nume)+1)];
    strcpy(m_nume, nume);
  }
  if (adresa != NULL) {
    m_adresa = new char [sizeof(char)*(strlen(adresa)+1)];
    strcpy(m_adresa, adresa);
  }
  m_matricol = matricol;
}

/* Constructor de copiere.
   Parametrii:
     s - referinta la un obiect de tip Student
*/
Student::Student(Student& s) {
  if (s.m_nume != NULL) {
    m_nume = new char [sizeof(char)*(strlen(s.m_nume)+1)];
    strcpy(m_nume, s.m_nume);
  }
  if (s.m_adresa != NULL) {
    m_adresa = new char [sizeof(char)*(strlen(s.m_adresa)+1)];
```

```
    strcpy(m_adresa, s.m_adresa);
  }

  m_matricol = s.m_matricol;
}

/* Elibereaza zona de memorie referita de variabilele m_nume si m_adresa.
Este apelata de destructor.
*/
void Student::makefree() {
  if (m_nume != NULL)
    delete[] m_nume;
  if (m_adresa != NULL)
    delete[] m_adresa;
}

/*Destructorul clasei Student*/
Student::~Student() {
  makefree();
}

/*Afiseaza intr-un mod cat mai placut elementele componente ale clasei.*/
void Student::afis() {
  cout<<"Nume: " << m_nume << "\t";
  cout<<"Adresa: " << m_adresa << "\t";
  cout<<"Matricol: " << m_matricol << endl;
}

/* Functie de tip "getter": intoarce valoarea variabilei interne m_nume.*/
char* Student::getName() {
  return m_nume;
}
```

========== Testree.cpp ==========

```
#include "binstree.h"
#include "student.h"

/* TODO:
   Sa se scrie o functie care sa:
     - introduca nodurile arborelui
     - stearga noduri pe baza unei chei
*/

Student* clasa[] = {new Student("Maria","A",1),new Student("Alin","B",2),
                new Student("Traian","C",3), new Student("Decebal","D",4),
                new Student("Cristina","E",5),new Student("Miruna","F",6),
                new Student("Maria","G",7),new Student("Dragos","H",8),
                new Student("Daniel","I",9),new Student("Costica","J",10),
                new Student("Daniela","K",11),new Student("Constantin","L",12),
                new Student("Dorin","M",13),new Student("Florica","N",14)
            };

void main(void) {
  BinSTree bst;
  char key[50];

  for (int i = 0; i < 14; i++)
    bst.insert(clasa[i]->getName(), clasa[i]);

  bst.inordine();
  while (1) {
```

```
    cout << "Cheia: ";
    cin >> key;
    if (strcmp(key, "Exit") == 0)
      break;
    bst.remove(key);
    bst.inordine();
  }
}
```

Temă

Bibliografie

http://inf.ucv.ro/~mirel/courses/poo/poo.html

1. *The C++ Programming Language*, B. Stroustrup, Addison-Wesley. (Cap. 13 Templates)