MEMORY ALLOCATOR

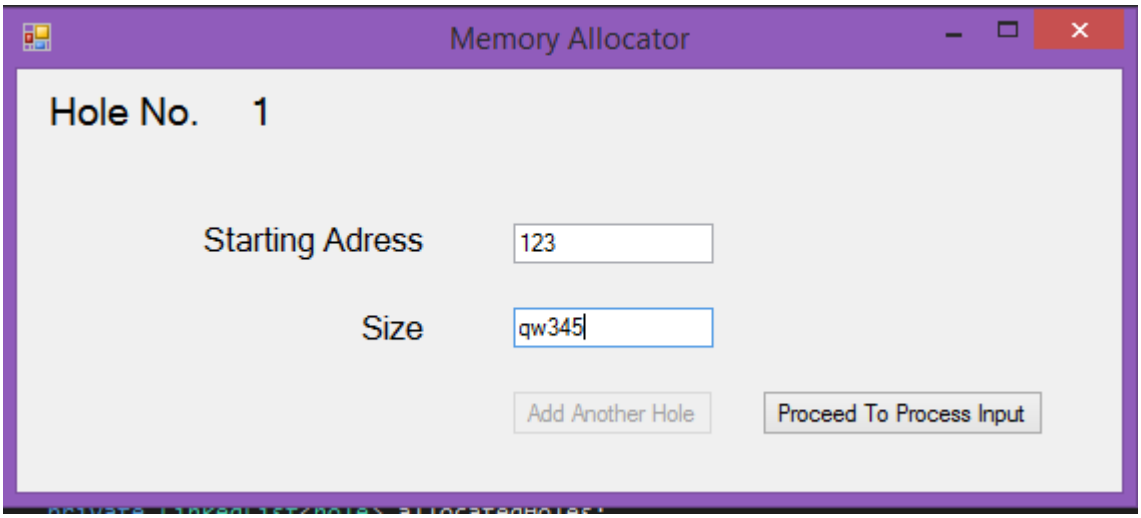
OS assignment CSE 15

SUBMITTED BY:AMR MOAHMED AHMED MAHMOUD, SEC: 2
SUBMITTED TO: ENG/ MAY MOHAMED

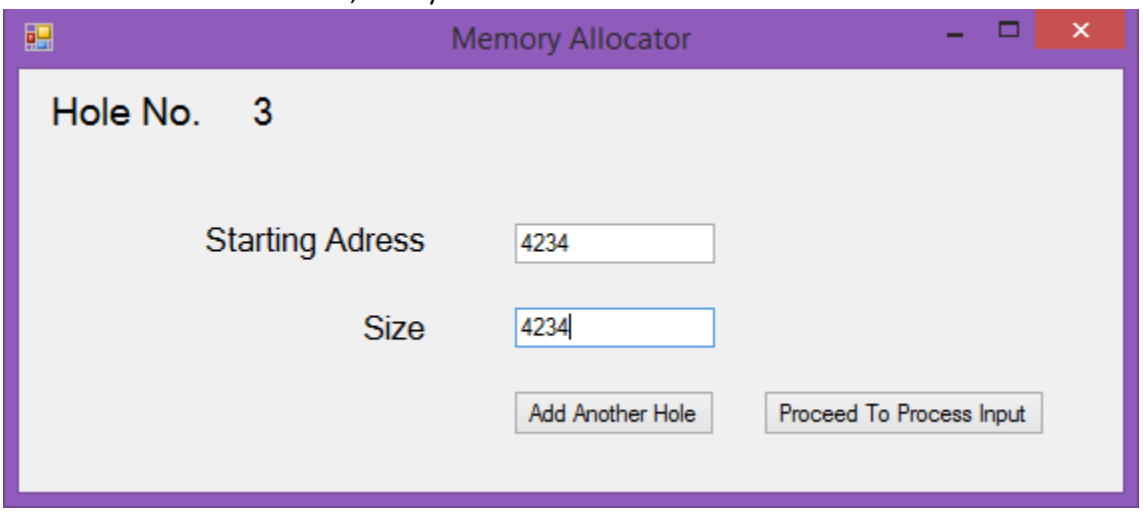
Manual

1. Entering the holes

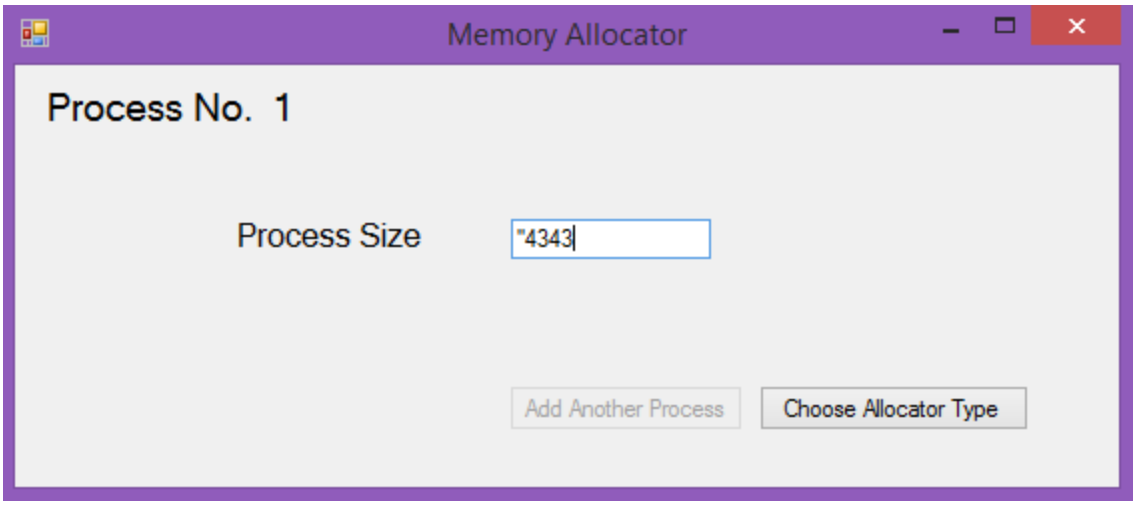
- Open the .exe file
- Enter a holes' start address and size
- Note that the button" Add Another Hole" will always be disabled until both start and size field contain valid positive integer numbers



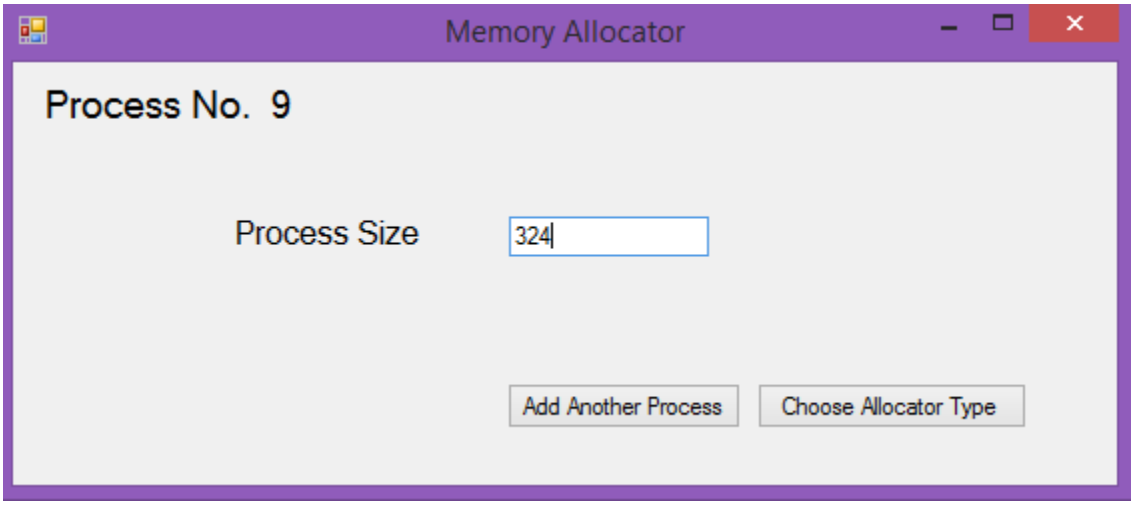
- When you are done entering the hole (note that you can enter as many hole as you want) press "Process to process input" to start entering processes; note that "proceed to process input also reads the start and size fields, if they contain valid values a hole will be added with these values



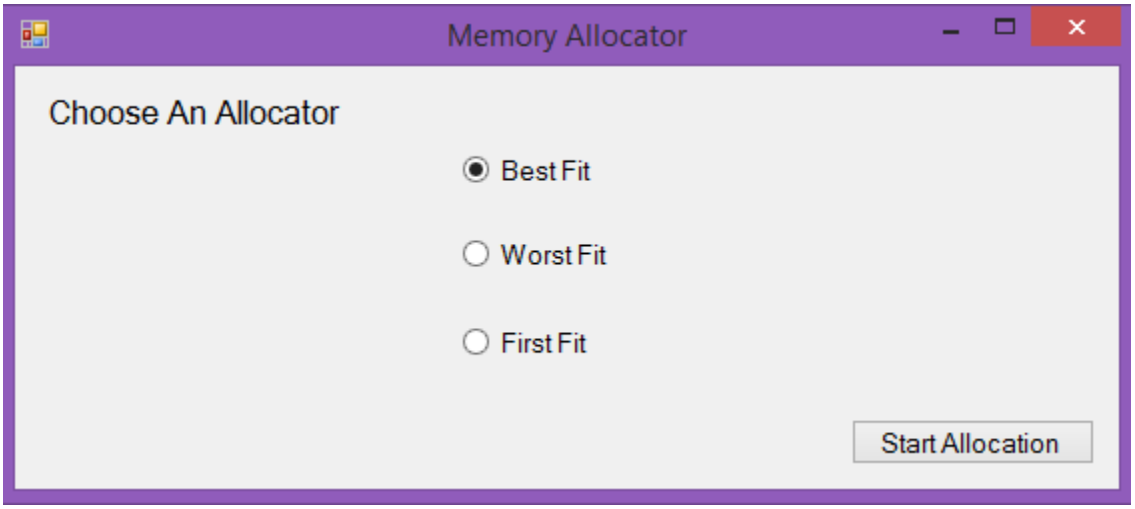
- Start entering processes' sizes
- Note that also the button "Add Another Process" will be disabled until the size field contain a valid positive integer only



- Enter as many process as you want(no limit on process numbers) then press “choose allocator type ” button; also note that this button reads the process size field and if iy contain a valid int a process will be added of that size



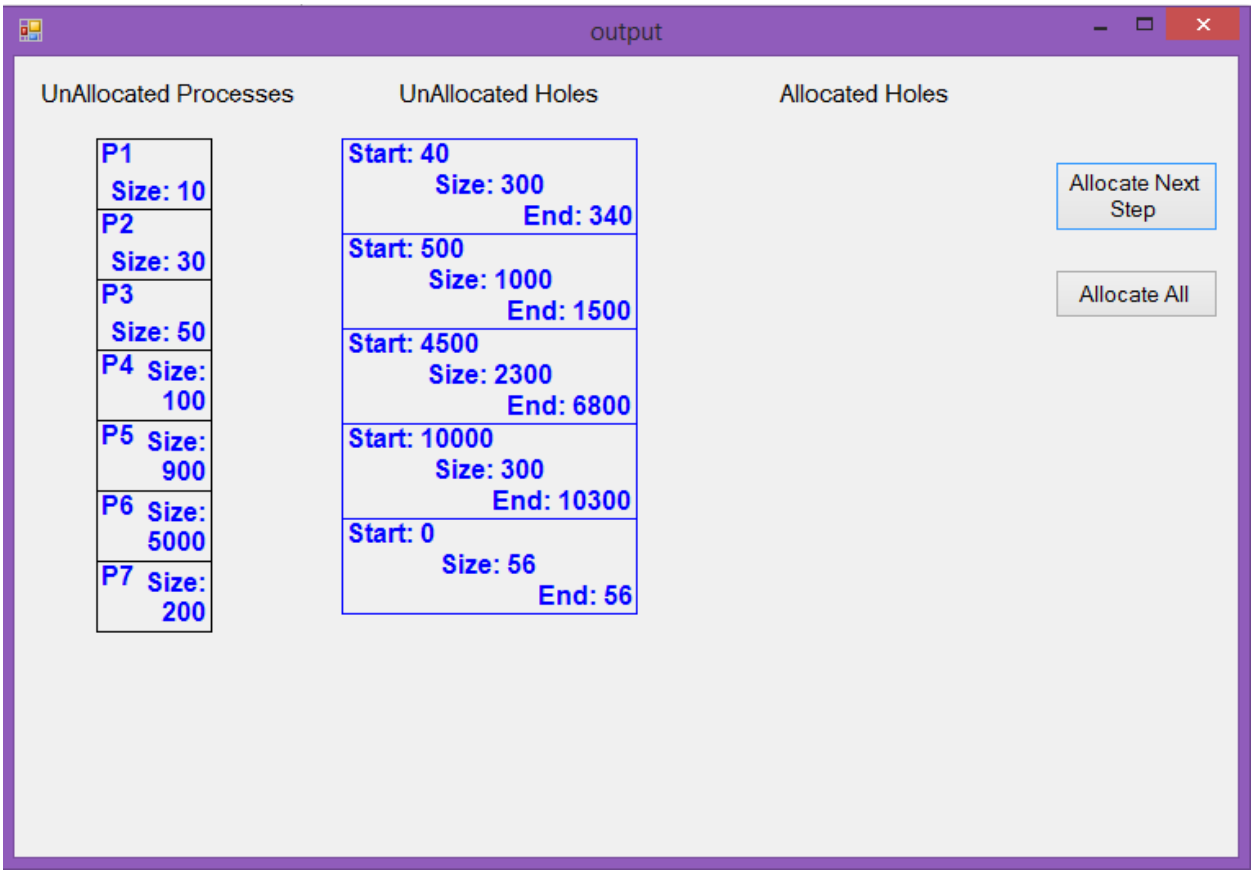
- Choose the type of allocator you want, then press “Start Allocation” button



- The following screenshot was taken after the following test case was entered allocator type "Best Fit"

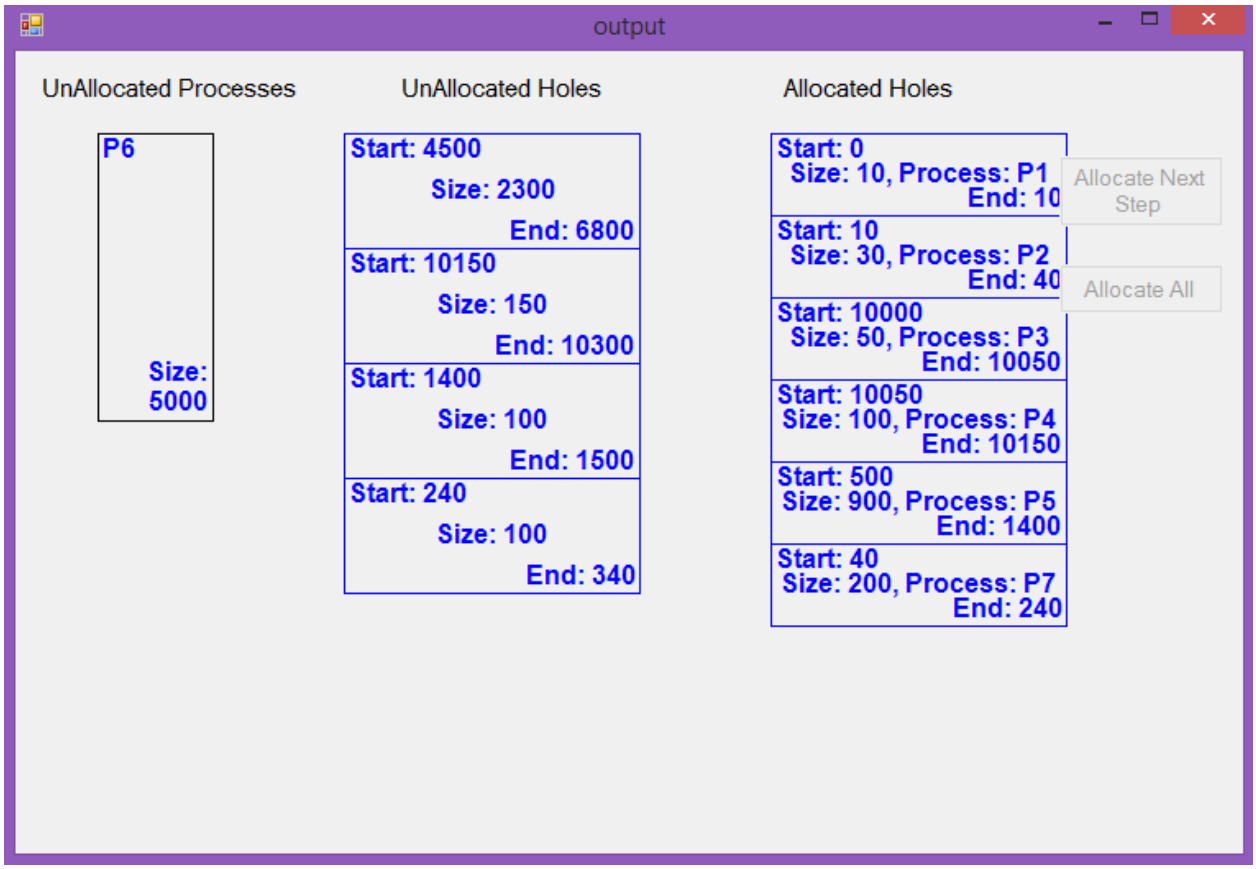
Hole number	Start	Size
1	0	10
2	6	50
3	40	300
4	500	1000
5	4500	2300
6	10000	300

Process number	Size
1	10
2	30
3	50
4	100
5	900
6	5000
7	200



- Note that the holes that can be merged are automatically merged at every step of the program

- Now we have 2 choices “Allocate All” and “Allocate next Step”
  “Allocate all” : view final state of hole and processes and disable the 2 buttons
  “Allocate next step”: view state after allocation of one process and disable the 2 button when there are no processes left
- What follows is the output from the allocate all button



- Process p6 can’t be allocated as no hole is that big
- To enter another simulation you have to restart the app.

Classes

hole

Data

`private int start;`
`private int size;`
`string process;`

`public hole()`
`public hole(int x, int y)`
`public hole(int x, int y,string p)`
`public int getStart()`
`public int getSize ()`
`public void setSize(int s)`
`public void setStart(int s)`
`public String getProcess()`
`public void setProcess(string p)`

Process

Data

`private int order;`
`private int size;`
`private bool allocated`

`public process()`
`public process(int x, int y)`
`public process (int x, int y,bool a)`
`public int getOrder()`
`public int getSize()`
`public void setOrder(int x)`
`public void setSize(int x)`
`public void setAsAllocated()`
`public bool isAllocated()`

allocator

Data

`private int order;`
`private int size;`
`private bool allocated`

`public allocator()`
`{`
`public void addHole(hole h)`
`public void addProcess(process p)`
`public int getNumberOfHoles()`
`public int getNumberOfProcesses()`

```
public void refreshHoles(LinkedList <hole>h)
public LinkedList<hole> getHoles()
public LinkedList<process> getProcesses()
public LinkedList<hole> getAllocatedHoles()
public LinkedList<process> getUnAllocatedProcesses()
public void setType(string s)
public string getType() { return type; }
public void firstFitAllocateProcess(process p)
public void bestFitAllocateProcess(process p)
public void worstFitAllocateProcess(process p)
public void firstFit()
public void bestFit()
public void worstFit()
```

Variables

Data
`public static allocator alloc;`

Code Snippets

```
Allocator::public void bestFitAllocateProcess(process p)
        {
            if (variables.alloc.getHoles().Count() == 0)
            {

                unallocatedprocesses.AddLast(p);
                processes.Remove(p);

            }
            if (variables.alloc.getHoles().Count() > 0) this.refreshHoles(holes);
            hole bestFit = holes.ElementAt(0);
            bool matchFound = false;
            int bestFitIndex = 0;
            for (int i = 0; i < holes.Count(); i++)
            {
                if (holes.ElementAt(i).getSize() >= p.getSize())
                {

                    p.setAsAllocated();
                    processes.Remove(p);
                    if ((holes.ElementAt(i).getSize() < bestFit.getSize() &&
holes.ElementAt(i).getSize() >=
p.getSize()&&matchFound)||(holes.ElementAt(i).getSize()>=p.getSize()&&!matchFound))
                    {
                        bestFit = holes.ElementAt(i);
                        bestFitIndex = i;
                    }
                    matchFound = true;
                }
            }
            if (matchFound)
            {
                if (bestFit.getSize() == p.getSize())
                {
                    holes.ElementAt(bestFitIndex).setProcess("P" +
p.getOrder().ToString());
                    allocatedHoles.AddLast(holes.ElementAt(bestFitIndex));
                    holes.Remove(holes.ElementAt(bestFitIndex));
                }
                else if (bestFit.getSize()>p.getSize())
                {
                    allocatedHoles.AddLast(new hole(bestFit.getStart(), p.getSize(),
"P" + p.getOrder().ToString()));
                    hole temp=holes.ElementAt(bestFitIndex);
                    holes.Remove(holes.ElementAt(bestFitIndex));
                    holes.AddLast(new hole(bestFit.getStart() + p.getSize(),
bestFit.getSize() - p.getSize()));

                }

            }
            else
            {
                unallocatedprocesses.AddLast(p);
                processes.Remove(p);
```

```
            }

        }
Allocator:: public void bestFit()
        {
            if (variables.alloc.getHoles().Count() > 0) this.refreshHoles(holes);
            for (int i = 0; i < processes.Count(); i++)
            {
                bestFitAllocateProcess(processes.ElementAt(i));
                this.refreshHoles(holes);
                variables.alloc.bestFit();
                break;
            }
        }
```

```
Allocator::public void firstFitAllocateProcess(process p)
        {
            if (variables.alloc.getHoles().Count() == 0)
            {

                    unallocatedprocesses.AddLast(p);
                    processes.Remove(p);


            }
          if (variables.alloc.getHoles().Count()>0)  this.refreshHoles(holes);
            for (int i = 0; i < holes.Count(); i++)
            {
                if (holes.ElementAt(i).getSize() >= p.getSize())
                {
                    p.setAsAllocated();
                    processes.Remove(p);
                    if (holes.ElementAt(i).getSize() == p.getSize())
                    {
                        allocatedHoles.AddLast(new hole(holes.ElementAt(i).getStart(),
holes.ElementAt(i).getSize(), "P " + p.getOrder().ToString()));
                        holes.Remove(holes.ElementAt(i));
                        break;
                    }
                    else
                    {
                        hole temp = holes.ElementAt(i);
                        hole unallocatedhole = new hole(temp.getStart() + p.getSize(),
temp.getSize() - p.getSize());
                        holes.Remove(holes.ElementAt(i));
                        allocatedHoles.AddLast(new hole(temp.getStart(), p.getSize(),
"P " + p.getOrder().ToString()));
                        break;
                    }

                }
                else
                {
                    unallocatedprocesses.AddLast(p);
                    processes.Remove(p);
                }
            }
        }
```

```
Allocator:: public void firstFit()
        {
            if (variables.alloc.getHoles().Count() > 0) this.refreshHoles(holes);

            for (int i=0;i<processes.Count();i++)
                {
                    firstFitAllocateProcess(processes.ElementAt(i));
                    this.refreshHoles(holes);
                    variables.alloc.firstFit();
                    break;
                }
        }
```

```
       Allocator:: public void worstFitAllocateProcess(process p)
        {
            if (variables.alloc.getHoles().Count() == 0)
            {

                unallocatedprocesses.AddLast(p);
                processes.Remove(p);

            }
            if (variables.alloc.getHoles().Count() > 0) this.refreshHoles(holes);
            hole worstFit = holes.ElementAt(0);
            bool matchFound = false;
            int worstFitIndex = 0;
            for (int i = 0; i < holes.Count(); i++)
            {
                if (holes.ElementAt(i).getSize() >= p.getSize())
                {

                    p.setAsAllocated();
                    processes.Remove(p);
                    if ((holes.ElementAt(i).getSize() > worstFit.getSize() &&
holes.ElementAt(i).getSize() >= p.getSize() && matchFound) ||
(holes.ElementAt(i).getSize() >= p.getSize() && !matchFound))
                    {
                        worstFit = holes.ElementAt(i);
                        worstFitIndex = i;
                    }
                    matchFound = true;
                }
            }
            if (matchFound)
            {
                if (worstFit.getSize() == p.getSize())
                {
                    holes.ElementAt(worstFitIndex).setProcess("P" +
p.getOrder().ToString());
                    allocatedHoles.AddLast(holes.ElementAt(worstFitIndex));
                    holes.Remove(holes.ElementAt(worstFitIndex));
                }
                else if (worstFit.getSize() > p.getSize())
                {
                    allocatedHoles.AddLast(new hole(worstFit.getStart(), p.getSize(),
"P" + p.getOrder().ToString()));
                    hole temp = holes.ElementAt(worstFitIndex);
```

```
                    holes.Remove(holes.ElementAt(worstFitIndex));
                    holes.AddLast(new hole(worstFit.getStart() + p.getSize(),
worstFit.getSize() - p.getSize()));

                }

            }
            else
            {
                unallocatedprocesses.AddLast(p);
                processes.Remove(p);
            }

        }
Allocator:: public void worstFit()
        {
            if (variables.alloc.getHoles().Count() > 0) this.refreshHoles(holes);
            for (int i = 0; i < processes.Count(); i++)
            {
                worstFitAllocateProcess(processes.ElementAt(i));
                this.refreshHoles(holes);
                variables.alloc.worstFit();
                break;
            }
        }
```

```
Allocator:: public void refreshHoles(LinkedList <hole>h)// do possible merges between
//holes
        {
            for (int i = 0; i < h.Count(); i++)
            {

                for (int j = 0; j < h.Count(); j++)
                {
                    if (i == j) ;
                    else
                    {
                        if (h.ElementAt(i).getStart() + h.ElementAt(i).getSize() >=
h.ElementAt(j).getStart() && h.ElementAt(i).getStart() <= h.ElementAt(j).getStart())
                        {
                            hole temp = new hole();
                            hole removable = h.ElementAt(j);// h.Remove(h.ElementAt((i
< j) ? j - 1 : j));
                            temp.setStart((h.ElementAt(i).getStart() <
h.ElementAt(j).getStart()) ? h.ElementAt(i).getStart() : h.ElementAt(j).getStart());
                            temp.setSize((h.ElementAt(i).getStart() +
h.ElementAt(i).getSize() > h.ElementAt(j).getStart() + h.ElementAt(j).getSize()) ?
h.ElementAt(i).getStart() + h.ElementAt(i).getSize() - temp.getStart() :
h.ElementAt(j).getStart() + h.ElementAt(j).getSize() - temp.getStart());
                            h.Remove(h.ElementAt(i));
                            h.Remove(removable);
                            h.AddLast(temp);
                            variables.alloc.refreshHoles(h);
                            break;
                        }
                    }
```

```
                } break;
            }

        }
```