Lehrstuhl für Informatik III
Prof. Dr.-Ing. M. Nagl

Klausur
Grundgebiete der Informatik 2

(DPO98/04)

Datum: 19. 07. 2007

Algorithmen und Programmiertechniken

Name: *(in Druckschrift)*	
Matr. Nr.:	
Unterschrift	

Aufgabe	Max. Punkte	Korrektur		Einsichtnahme	
		Punkte	Kürzel	Punkte	Kürzel
1	9				
2	11				
3	20				
4	20				
Σ	60				

Aufgabe 1 C++-Syntax und -Semantik *(9 Punkte)*

(a) In der Funktion `main` werden die Variablen `x`, `y` und `z` als Aktualparameter an die Prozedur `f` übergeben. Geben Sie rechts in der Tabelle die Werte von `x`, `y` und `*z` nach jedem Aufruf der Prozedur `f` an. *(6 Punkte)*

```
void f(int a, int &b, int *c){
   a = b * (*c);
   b = a - (*c);
   (*c) = a + b;
}

int main(){
   int x = 5;
   int y = 2;
   int *z;
   (*z) = 1;
   f(x, y, z); // 1. Aufruf der Prozedur f
   // Geben Sie die Werte von x, y und z an
   f(x, y, z); // 2. Aufruf der Prozedur f
   // Geben Sie die Werte von x, y und z an
   return 0;
}
```

	x	y	*z
1. Prozeduraufruf	5	1	3
2. Prozeduraufruf	5	0	3

(b) Bei der Syntax einer Programmiersprache wird zwischen der lexikalischen, kontextfreien und kontextsensitiven Syntax unterschieden. Geben Sie für das Programm aus Teilaufgabe (a) jeweils ein Beispiel für jede der drei Syntaxebenen an. *(3 Punkte)*

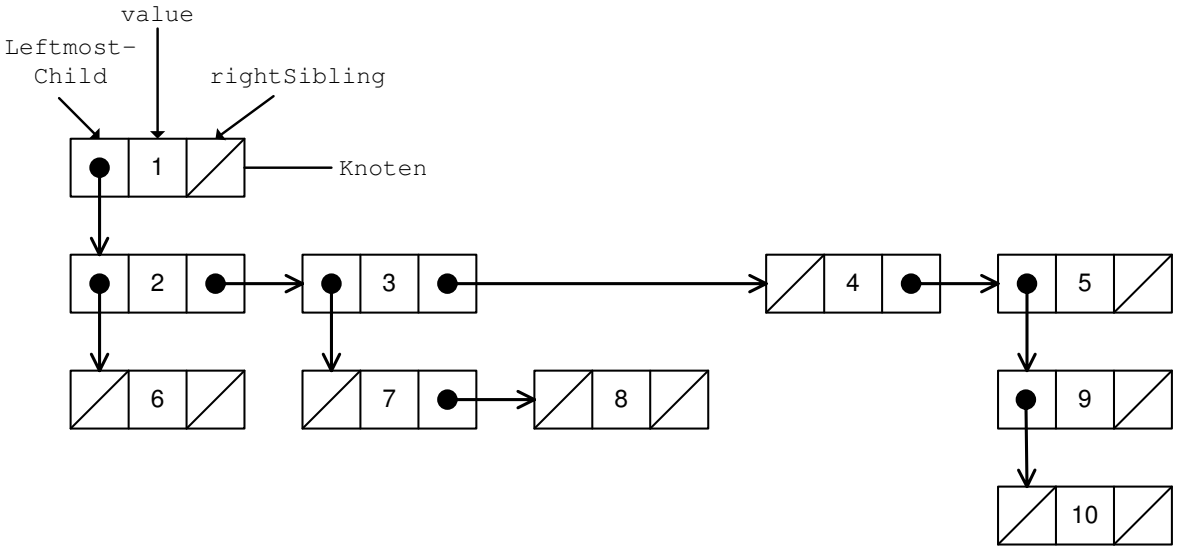
- lexikalische Syntax: z.B. Literal `5`
- kontextfreie Syntax: z.B. Initialisierte Deklaration `int x = 5;`
- kontextsensitive Syntax: z.B. `f(x, y, z)` ist eine Anwendungsstelle der Deklaration `void f(int a, int &b, int *c)`

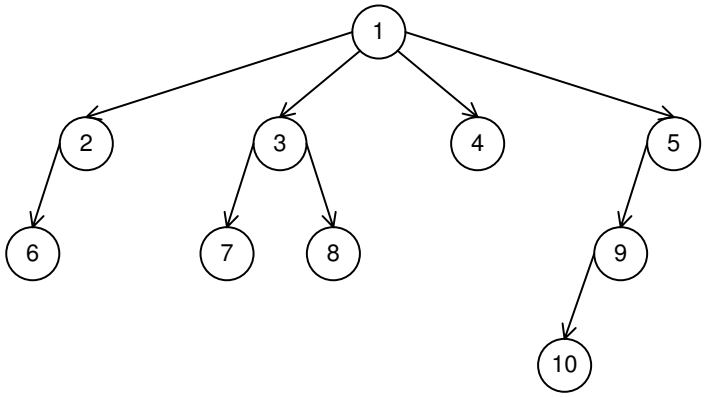
Aufgabe 2 Bäume und Graphen *(11 Punkte)*

In dieser Aufgabe soll die Speicherung von Bäumen und Graphen untersucht werden.

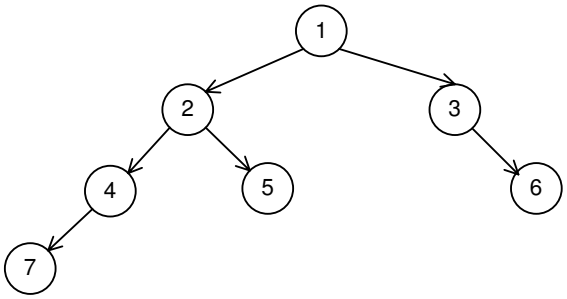
(a) Mittels der leftmostChild-rightSibling-Darstellung können n-äre Bäume gespeichert werden. Jeder Knoten des Baums wird in ein Listenelement des Verbundtyps Knoten gespeichert, der aus drei Komponenten besteht (siehe nachfolgende Typdefinition Knoten). Betrachten Sie die folgende Abbildung: Konstruieren Sie aus dieser Darstellung den zugehörigen n-ären Baum und zeichnen Sie diesen unter die Abbildung. Schreiben Sie hierbei in jeden Knoten des Baums den Wert seiner value-Komponente. *(4 Punkte)*

```
struct Knoten {
   Knoten *leftmostChild;
   int value;
   Knoten *rightSibling;
}
```



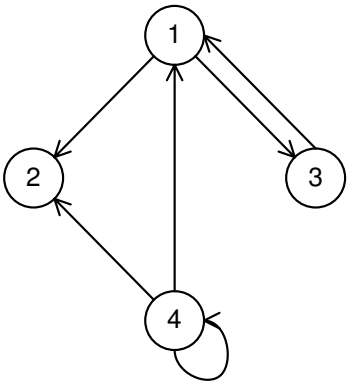


(b) Geben Sie die Werte, die in folgendem weiteren binären Baum gespeichert sind, in Postorder-Reihenfolge an. *(3 Punkte)*



7 4 5 2 6 3 1

(c) Geben Sie für den folgenden gerichteten Graphen die Speicherung als sequentielle Adjazenzliste an. Füllen Sie hierzu die Datenstrukturen `NeighPlus` und `Last` in der Abbildung aus. Im Feld `NeighPlus` werden in Zeile `i` die Nachfolgerknoten von Knoten `i` gespeichert, die also über eine auslaufende Kante erreicht werden können. `Last` speichert für jeden Knoten den jeweiligen Spaltenindex der Stelle in `NeighPlus`, an der der *letzte* Nachfolger des Knotens steht. Die Nachfolgerknoten sollen im Feld `NeighPlus` geordnet sein. *(4 Punkte)*



Graph

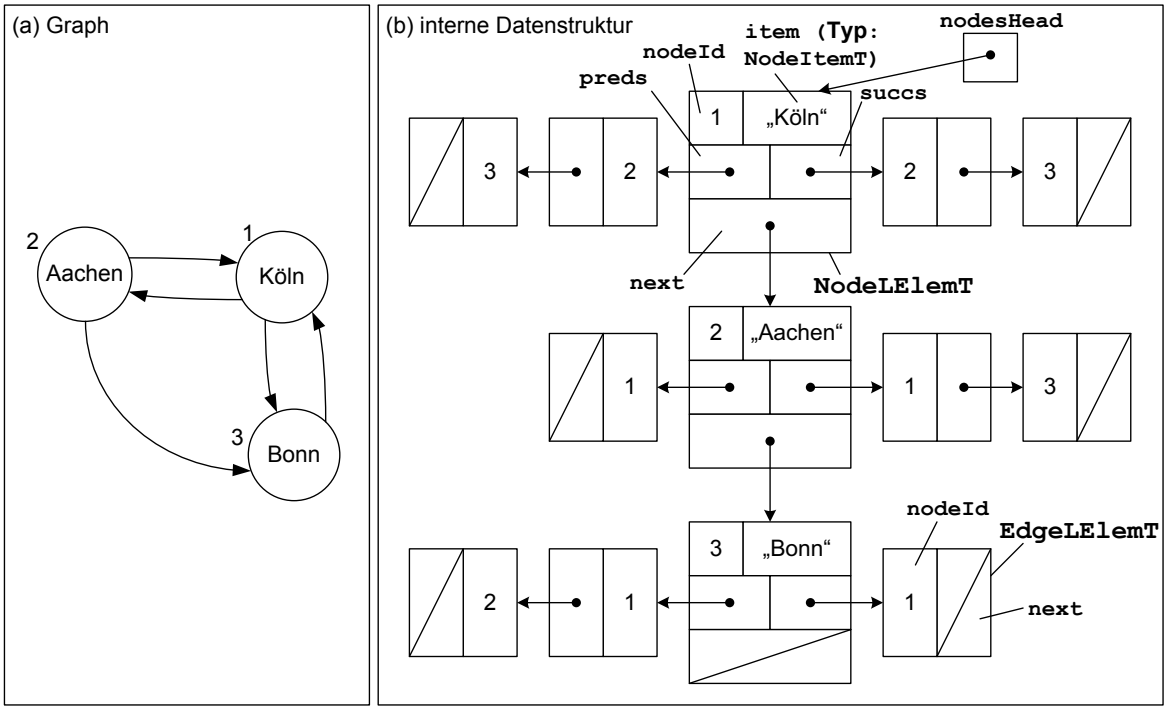
	NeighPlus				Last
1	2	3			2
2					0
3	1				1
4	1	2	4		3

Aufgabe 3 — ADT Graph — *(20 Punkte)*

In dieser Aufgabe soll ein gerichteter Graph als Klasse implementiert werden. Führt eine gerichtete Kante von Knoten i zu Knoten j, so nennt man j einen *Nachfolger* von i und i einen *Vorgänger* von j.

Die Knoten des Graphen können Informationen aufnehmen. Wir erstellen eine generische Lösung, damit später verschiedene Informationen aufgenommen werden können. Der Typ dieser Informationen ist somit der generische Typ `NodeItemT`. Die Klasse besitzt Operationen zur Manipulation der Graphstruktur und zur Iteration über Vorgänger- und Nachfolgerknoten eines angegebenen Knotens (s. Schnittstelle auf nächster Seite). Die Verwaltung der Knotenbezeichner (Typ `int`) obliegt dem Verwender des ADT `Graph`.

Intern wird der Graph über einfach verkettete Knoten- und Kantenlisten (in Adjazenzdarstellung) gespeichert. Jedes Element (Typ: `NodeLElemT`) aus der Knotenliste verfügt u.a. über einen Zeiger `succs` auf eine Kantenliste mit Nachfolgerknoten aber auch über einen Zeiger `preds` auf eine Kantenliste mit Vorgängerknoten. Ein Element (Typ: `EdgeLElemT`) aus einer Kantenliste speichert u.a. den Knotenbezeichner des Nachfolger- bzw. Vorgängerknotens.



Die obige Abbildung zeigt (a) ein Beispiel für einen Graphen und (b) seine interne Realisierung. Der generische Typ `NodeItemT` der Knoteninformationen ist hier als `string` gewählt.

```
template <class NodeItemT>
class Graph
{
public:
    // Fügt einen Knoten mit der Knoteninformation 'item' ein. Der
    // Rückgabewert ist der neue, eindeutige, ganzzahlige Knotenbezeichner.
    int AddNode(NodeItemT item);

    // Fügt Kante von Knoten 'from' zu Knoten 'to' ein.
    void AddEdge(int from, int to);

    // Entfernt Kante von Knoten 'from' zu 'to'.
    void RemoveEdge(int from, int to);

    // Gibt genau dann 'true' zurück, wenn Knoten 'nodeId' einen
    // Nachfolger besitzt, der dann in 'succNode' zurückgegeben wird.
    bool GetFirstSucc(int nodeId, int& succNode);

    // Gibt genau dann 'true' zurück, wenn ein weiterer Nachfolger-
    // knoten existiert, der dann in 'succNode' zurückgeliefert wird.
    bool GetNextSucc(int& succNode);

    // Beide Funktionen verhalten sich analog zu den beiden vorher-
    // gehenden, dienen aber zur Iterationen über Vorgänger-Knoten.
    bool GetFirstPred(int nodeId, int& predNode);
    bool GetNextPred(int& predNode);

private:
    // Deklarationen für die interne Datenstruktur (s. Aufgabenteil a)
    // ...


    // Speichert den zuletzt vergebenen Knotenbezeichner
    int lastNodeId;

    // Zeiger für Iteration über Kanten
    EdgeLElemT* cursor;

    // Löscht Element mit 'nodeId' aus Kantenliste, auf deren
    // Kopf 'edgesHead' verweist.
    void DeleteFromEdgeList(EdgeLElemT* & edgesHead, int nodeId);

    // Gibt Zeiger auf Element in Knotenliste mit
    // übergebener 'nodeId' zurück.
    NodeLElemT* GetNodeLElem(int nodeId);

};
```

(a) Vervollständigen Sie den privaten Teil der Schnittstelle von **Graph**. Geben Sie die Deklarationen von **NodeLElemT**, **EdgeLElemT** sowie **nodesHead** an (vgl. Teil (b) der vorangegangenen Abbildung). *(4 Punkte)*

```
struct EdgeLElemT { //Lx
    EdgeLElemT* next; //Lx
    int nodeId; //Lx
}; //Lx

struct NodeLElemT { //Lx
    int nodeId; //Lx
    NodeItemT item; //Lx
    EdgeLElemT* succs; //Lx
    EdgeLElemT* preds; //Lx
    NodeLElemT* next; //Lx
}; //Lx

NodeLElemT* nodesHead; //Lx
```

(b) Implementieren Sie die Prozedur **RemoveEdge**, die eine gerichtete Kante aus dem Graphen entfernt. Der Parameter **from** gibt den Bezeichner des Quellknotens an, der Parameter **to** den Bezeichner des Zielknotens. Sie können davon ausgehen, dass die Kante im Graphen existiert. *Hinweise:* Verwenden Sie die Hilfsfunktionen **GetNodeLElem** und **DeleteFromEdgeList**. *(6 Punkte)*

```
template <class NodeItemT>
void Graph<NodeItemT>::RemoveEdge(int from, int to)
{
    DeleteFromEdgeList(GetNodeLElem(from)->succs, to); //Lx
    DeleteFromEdgeList(GetNodeLElem(to)->preds, from); //Lx




}
```

(c) Implementieren Sie die Funktion AddNode. Dieser wird mit item die im Knoten zu speichernde Information übergeben. Der Rückgabewert ist ein bisher nicht vergebener, ganzzahliger Bezeichner für den neuen Knoten. *Hinweise:* Fügen Sie das neue Element am Anfang der Knotenliste ein. In lastNodeId im privaten Teil der Schnittstelle wird der zuletzt vergebene Knotenbezeichner gespeichert.
(6 Punkte)

```
template <class NodeItemT>
int Graph<NodeItemT>::AddNode(NodeItemT item)
{
    lastNodeId++; //Lx

    NodeLElemT* listElem = new NodeLElemT; //Lx
    listElem->nodeId = lastNodeId; //Lx
    listElem->item = item;  //Lx
    listElem->next = nodesHead; //Lx
    listElem->preds = NULL; //Lx
    listElem->succs = NULL; //Lx
    nodesHead = listElem; //Lx

    return lastNodeId; //Lx

}
```

(d) Beschreiben Sie kurz und in Stichpunkten (kein Programmcode), wie die interne Datenstruktur beim Löschen eines Knotens aus dem Graph behandelt werden muss. *(4 Punkte)*

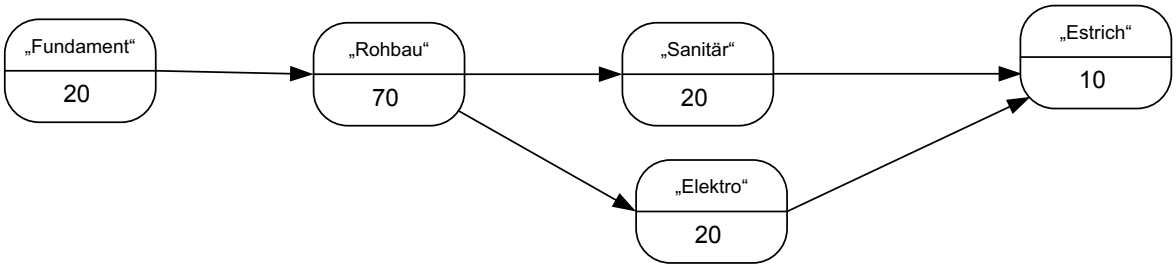
- Löschen des betreffenden Elements aus der Knotenliste
- Löschen der dem gelöschten Knotenelement anhängenden Kantenlisten
- Iteration durch Knotenliste
- und dabei Löschen aller Kantenlistenelemente, die auf gelöschten Knoten verweisen.

Aufgabe 4 ADT Tasknet *(20 Punkte)*

In dieser Aufgabe soll ein ADT Tasknet zur Verwaltung eines Aufgabennetzes implementiert werden. Ein Aufgabennetz verwaltet Informationen (Aufgabenname und -dauer) für eine Menge von Aufgaben und die Abhängigkeiten zwischen verschiedenen Aufgaben.

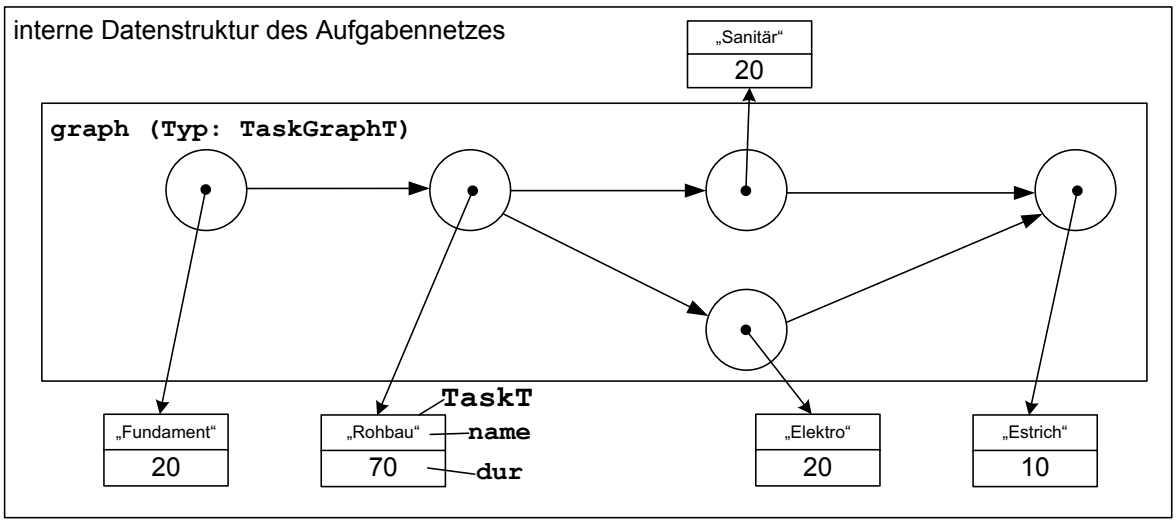
```
// Typdefinition für die Aufgabenbezeichner
typedef int TaskId;

class Tasknet
{
public:

    // Legt eine neue, isolierte Aufgabe mit Namen 'name' und Dauer 'dur'
    // im Aufgabennetz an. Der Rückgabewert ist ein eindeutiger
    // Bezeichner.
    TaskId AddTask(string name, int dur);

    // Macht aus isolierter Aufgabe 'succTask' eine Nachfolgeraufgabe der
    // Aufgabe 'task'. Die ursprünglichen Nachfolger von 'task'
    // sind anschließend nur noch Nachfolger von 'succTask'.
    void AddSuccTask(TaskId task, TaskId succTask);

    // Macht aus isolierter Aufgabe 'parTask' eine zu 'task' parallele
    // Aufgabe. 'parTask' erhält die gleichen Vorgänger- und
    // Nachfolgeraufgaben wie 'task'.
    void AddParTask(TaskId task, TaskId parTask);

private:
    // Deklarationen (s. Aufgabenteil b)
    // ...


};
```

(a) Vervollständigen Sie die Implementierung von CreateSampleTasknet. Nach Abarbeitung dieser Funktion soll das Aufgabennetz t so aussehen wie in der Abbildung dargestellt. *(4 Punkte)*




```
void CreateSampleTasknet() {
    Tasknet t;
    TaskId fund = t.AddTask("Fundament", 20);
    TaskId rohb = t.AddTask("Rohbau", 70); //Lx
    TaskId sani = t.AddTask("Sanitär", 10); //Lx
    TaskId elek = t.AddTask("Elektro", 10);//Lx
    TaskId estr = t.AddTask("Estrich", 8);//Lx

    t.AddSuccTask(fund, rohb);//Lx
    t.AddSuccTask(rohb, sani);//Lx
    t.AddSuccTask(sani, estr);//Lx

    t.AddParTask(sani, elek);//Lx









}
```

(b) Der ADT **Tasknet** baut auf dem ADT **Graph** der vorherigen Aufgabe auf. Die Knoteninformation ist in diesem Fall ein Zeiger auf ein Verbund aus Aufgabenname und -dauer. Die Abbildung zeigt die interne Datenstruktur für das vorige Aufgabennetz.



Vervollständigen Sie den privaten Teil der Schnittstelle von **Tasknet** in vier Schritten. Schreiben Sie die Lösung für jeden Schritt unter den zugehörigen Kommentar. *(5 Punkte)*

```
// 1. Definieren Sie einen Verbundtyp 'TaskT'.
struct TaskT {  //Lx
    string name; //Lx
    int dur; //Lx
}; //Lx
```

```
// 2. Definieren Sie einen Typ 'ZgrTaskT' für Zeiger auf
// Datenobjekte vom Typ 'TaskT'.
typedef TaskT* ZgrTaskT; //Lx
```

```
// 3. Definieren Sie einen Typ 'TaskGraphT' für Graphen mit
// Knoteninformationen vom Typ 'ZgrTaskT'.
// (Hinweis: generische Instanzerzeugung)
typedef Graph<ZgrTaskT> TaskGraphT; //Lx
```

```
// 4. Definieren Sie ein Datenobjekt 'graph' vom Typ 'TaskGraphT'.
TaskGraphT graph;
```

(c) Implementieren Sie die Funktion AddTask. Diese erzeugt einen neuen, zunächst isolierten Knoten im zugrundeliegenden Graph graph. Die Knoteninformation ist vom Typ ZgrTaskT (s. Aufgabenteil b), also ein Zeiger auf einen neuen Verbund vom Typ TaskT, in dem die Werte der Parameter gespeichert werden. *(5 Punkte)*

```
TaskId Tasknet::AddTask(string name, int dur)
{
    ZgrTaskT newTask = new TaskT; //Lx
    newTask->name = name; //Lx
    newTask->dur = dur; //Lx
    return graph.AddNode(newTask); //Lx




}
```

(d) Implementieren Sie die Prozedur AddSuccTask. Diese fügt als Nachfolger von task eine bereits existierende Aufgabe succTask ein. Alle ursprünglichen Nachfolger von task sind anschließend Nachfolger von succTask aber nicht mehr von task. *Hinweis:* Verwenden Sie hier insbesondere die Funktionen GetFirstSucc und GetNextSucc aus der Schnittstelle des ADT Graph. *(6 Punkte)*

```
void Tasknet::AddSuccTask(TaskId task, TaskId succTask)
{
    TaskId oldSucc; //Lx
    if (graph.GetFirstSucc(task, oldSucc)) { //Lx
        graph.RemoveEdge(task, oldSucc); //Lx
        graph.AddEdge(succTask, oldSucc); //Lx
        while (graph.GetNextSucc(oldSucc)) { //Lx
            graph.RemoveEdge(task, oldSucc); //Lx
            graph.AddEdge(succTask, oldSucc); //Lx
        } //Lx
    } //Lx

    graph.AddEdge(task, succTask); //Lx




}
```