CXX01

Graph library documentatie

Advanced C Programming

Rene Schouten (0928619)
Raber Ahmad (0921954)

Interne structuur

Een graph bestaat uit een gelinkte lijst van vertexes. Een vertex is een node in de graph.

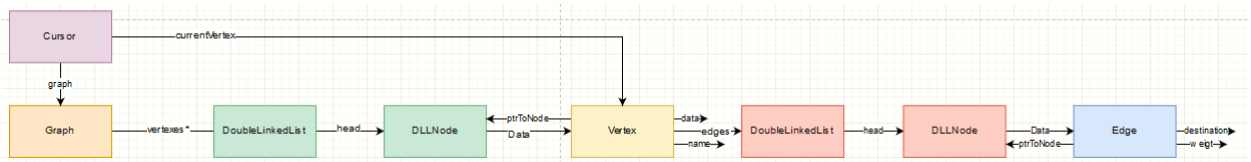
```
typedef struct
{
    DoublyLinkedList *vertices;
} Graph;
```

Een Vertex object heeft een naam, data van de gebruiker, en een lijst met edges, deze edges zijn verbindingen naar andere vertexes. Ook heeft de vertex een pointer naar de DLLNode van de doublelinkedlist dat de library nodig heeft voor het correct verwijderen van vertexes.

De naam van een vertex wordt bij de initialisatie van de vertex gekopieerd en de library is verantwoordelijk voor het vrijmaken van dat geheugen. De data van een vertex is eigendom van de gebruiker en zal niet door de graph library vrijgemaakt worden! Dit omdat het ook geheugen van de stack kan zijn

```
typedef struct
{
    char* name;
    void* data;
    DoublyLinkedList *edges;
    DllNode* ptrToNode;
} Vertex;
```

Een edge is een object dat verwijst naar een andere vertex(destination) en heeft een gewicht(weight) wanneer het gewicht niet gebruikt wordt is het gewicht 0. Ook de edge heeft een pointer naar de DLLNode zodat de library het object goed kan verwijderen.



API

```
Graph* createGraph(void);
```

Return een graph object, deze graph is nog helemaal leeg.

```
void graphDelete(Graph** ptrToDeleteGraph);
verwijderd de graph met alle vertexes en edges uit het geheugen en zet de
pointer op NULL. Let op de gebruikersdata word niet verwijderd.

Vertex* addVertex(Graph *graph, char* name, void* data);
//delete a vertex and set the pointer to NULL
void deleteVertex(Graph *graph,Vertex** ptrToDeleteVertex);

//returns the number of vertexes in a given graph
size_t numberOfVertexs(Graph* graph);

//creates a edge(connection) between vertexes, which can bo both directed or
undirected
Edge* createEdge(Vertex* from, Vertex* destination, bool directed);
```

```
//creates a edge(connection) between vertexes, which can bo both directed or
undirected. also a weight is added
Edge* createEdgeWithWeight(Vertex* from, Vertex* destination,int weight, bool
directed);

//delete a edge
void deleteEdge(Graph* graph, Edge *toDeleteEdge, Vertex* connectedVertex);

//print a vertex and its connections
void vertexPrintConnections(Graph* graph, Vertex* pointOfView);

//returns the first vertex with the given name, if it don't exist it will
return NULL
Vertex* searchVertexByName(Graph* graph, char* name);

//delete all vertexes and edges
void clear(Graph* graph);

//clear all vertexes that are not connected
void clearFloatingVertexes(Graph* graph);
```

De cursor library

Een gebruiker van de graph zou niet de interne structuur van de library met doublelinkedlists hoeven kennen, om zonder die kennis door de graph te navigeren is een cursor volgens het cursor design pattern gemaakt.

Een cursor staat op een bepaalde plek in de graph en er van vertex naar vertex door heen navigeren.

```
typedef struct
{
    Graph* graph;
    Vertex* currentVertex;
} GraphCursor;
```

Cursor API

```
GraphCursor* createCursor(void);
```
Maakt een cursor, deze cursor staat nog niet op een plek in de graph.

```
GraphCursor* copyGraphCursor(GraphCursor* cursor);
```
Kopieer een bestaande cursor.

```
void deleteCursor(GraphCursor** ptrToCursor);
```
verwijder een cursor en zet de pointer naar NULL.

```
void cursorSetCurrentVertex(GraphCursor *cursor, Graph* graph, Vertex*
vertex);
```
zet de cursor op een bepaalde vertex in een graph.

```
int cursorAvailable(GraphCursor* cursor);
```
returnt hoeveel edges de vertex heeft. Wanneer de cursor niet op een vertex staat return het 0.

```
Edge* cursorEdgeAt(GraphCursor* cursor, int index);
```

Return een edge op de zoveelste positive van de vertex. Wanneer het getal groter is dan cursorAvailable() word NULL gereturnt.

```
Vertex* cursorAt(GraphCursor* cursor, int index);
```
Return een vertex op de zoveelste positive van de vertex. Wanneer het getal groter is dan cursorAvailable() word NULL gereturnt.

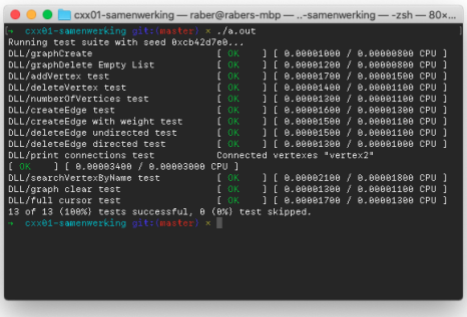
```
void cursorMoveInto(GraphCursor* cursor, int index);
```
zet de cursor zoveelste egde positie van de vertex.
```
Vertex* cursorGetCurrentVertex(GraphCursor* cursor);
```
Return de huidige vertex waar de cursor op staat
```
void* cursorGetCurrentData(GraphCursor* cursor);
```
Return de huidige data van de vertex waar de cursor op staat

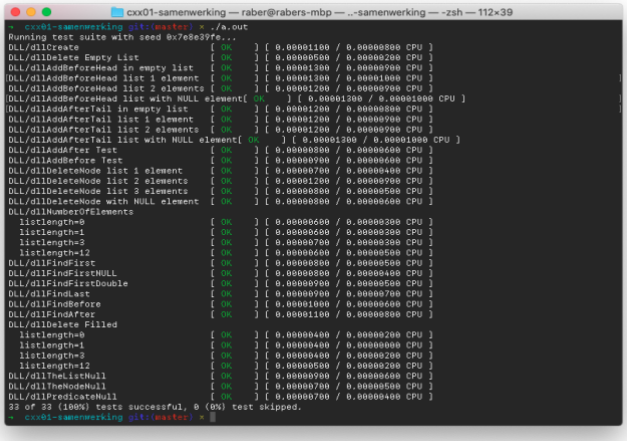
```
void cursorDeleteCurrentVertex(GraphCursor *cursor);
```
verwijderd de vertex waar de cursor op staat en zet de de cursor positive op NULL.
```
void cursorDeleteEdgeAt(GraphCursor *cursor, int index);
```

verwijderd een edge op een bepaalde index

Valgrind en Tests

Hieronder zien we alle testresultaten van de Graph als die van de doublelinkedlist



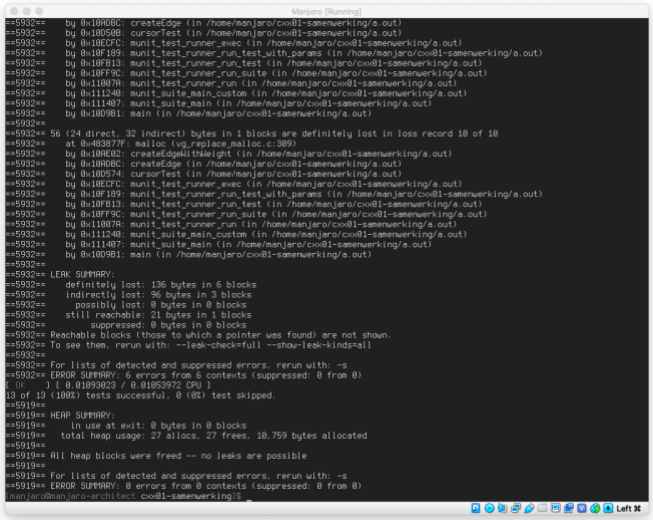


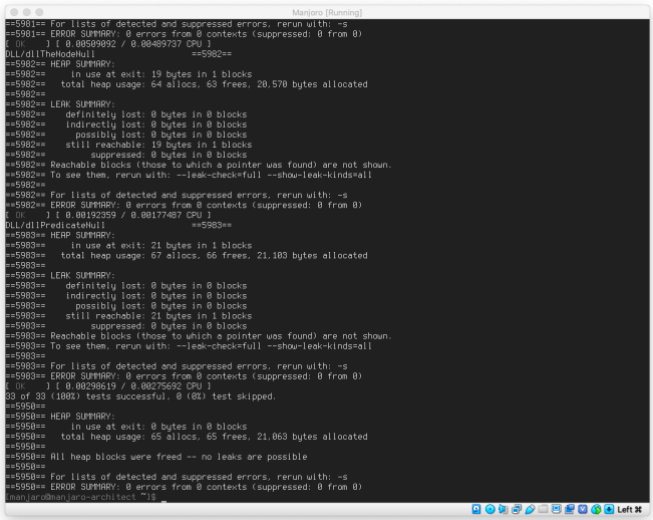
Linkst staan de testen van de graph en rechts staan de testen van de doublelinkedlist. Zoals te zien zijn beide testen 100% geslaagd.

De graph is op de volgende punten getest:

- NULL pointers
- Creëren van een graph
- Verwijderen van een graph
- Toevoegen en verwijderen van een vertex
- Retourneren van de juiste aantal Vertexes

- Creëren van een edge met en zonder weigt
- Verwijderen van een edge voor zwel directed als undirected





Hierboven zien we screenshot van de valgrind resultaten. De linker is het resultaat van de dllist+ graph en die van de rechter is het valgrind resultaat van alleen de dllist.

We kunnen zien dat de dll alleen geen memory-leaks alles wat wordt gealloceerd wordt ook weer verwijderd. We zien staan dat bij defintely lost, 0 bytes staan.

De graph samen met dll geeft wel aan dat er een memory leak is. We kunnen dat ook weer zien bij defintely lost in de linker afbeelding. Er staat namelijk dat er 136 bytes verloren zijn gegaan in 6 blokken.