HANYANG UNIVERSITY
DIVISION OF COMPUTER SCIENCE

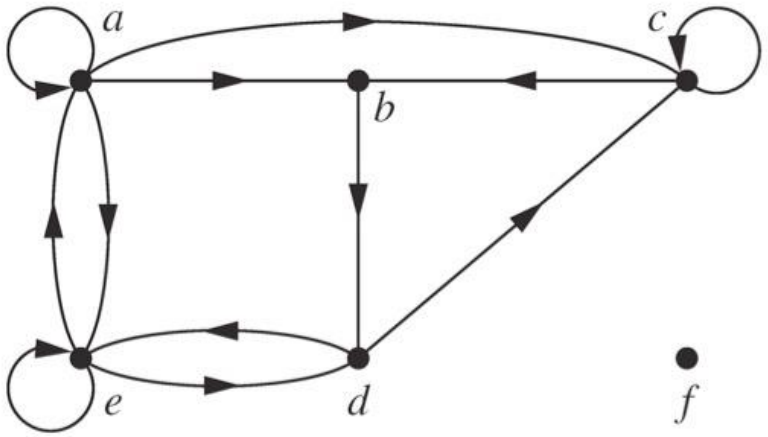
Data Structures

박영준 교수님

Lab10: Graph

Graph

- Non-linear data structure consisting of nodes(vertex) and edges.



SimpleGraph ADT

- void InitGraph(Graph *G, int NumofVertex);
  - Initiate graph
- void ReleaseGraph(Graph *G);
  - Release graph
- void AddEdge(Graph *G, int From, int To);
  - Link vertex From and To with edge
- void PrintGraph(Graph *G);
  - Print graph informations

SimpleGraph

```c
#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0

typedef int DATATYPE;

enum {A, B, C, D, E, F, G, H, I, J};

typedef struct Node{
    DATATYPE Data;
    struct Node *next;
} Node;

typedef struct LinkedList
{
    Node *Head;
    Node *Curr;
    Node *Prev;
    int NumofData;
} LinkedList;

typedef struct
{
    int NumofVertex;
    int NumofEdge;
    struct LinkedList *AdjList;
} Graph;

//list
void InitList(LinkedList *list);
void Insert(LinkedList *list, DATATYPE Data);

void HeadInsert(LinkedList *list, DATATYPE Data);
void SortInsert(LinkedList *list, DATATYPE Data);

int PosHead(LinkedList *list, DATATYPE *Data);
int PosNext(LinkedList *list, DATATYPE *Data);

DATATYPE Remove(LinkedList *list);
int RetCount(LinkedList *list);

//graph
void InitGraph(Graph *G, int NumofVertex);
void ReleaseGraph(Graph* G);

void AddEdge(Graph *G, int From, int To);
void PrintGraph(Graph *G);
```

SimpleGraph

```
70 //list
71 void InitList(LinkedList *list)
72 {
73     list->Head = (Node*)malloc(sizeof(Node));
74     list->Head->next = NULL;
75     list->NumofData = 0;
76 }
77
78 void Insert(LinkedList *list, DATATYPE Data)
79 {
80     if(list->Head->next == NULL)
81     {
82         HeadInsert(list, Data);
83     }
84     else
85     {
86         SortInsert(list, Data);
87     }
88 }
```

```
90 void HeadInsert(LinkedList *list, DATATYPE Data)
91 {
92     Node *temp = (Node*)malloc(sizeof(Node));
93     temp->Data = Data;
94
95     temp->next = list->Head->next;
96     list->Head->next = temp;
97
98     list->NumofData++;
99 }
100
101 void SortInsert(LinkedList *list, DATATYPE Data)
102 {
103     Node *new = (Node*)malloc(sizeof(Node));
104     Node *pred = list->Head;
105     new->Data = Data;
106
107     //find pos
108     while((pred->next != NULL) && (Data > pred->next->Data))
109     {
110         pred = pred->next;
111     }
112
113     new->next = pred->next;
114     pred->next = new;
115
116     list->NumofData++;
117 }
```

SimpleGraph

```
119 int PosHead(LinkedList *list, DATATYPE *Data)
120 {
121     if(list->Head->next == NULL)
122     {
123         return FALSE;
124     }
125
126     list->Prev = list->Head;
127     list->Curr = list->Head->next;
128
129     *Data = list->Curr->Data;
130     return TRUE;
131 }
132
133 int PosNext(LinkedList *list, DATATYPE *Data)
134 {
135     if(list->Curr->next == NULL)
136     {
137         return FALSE;
138     }
139
140     list->Prev = list->Curr;
141     list->Curr = list->Curr->next;
142
143     *Data = list->Curr->Data;
144     return TRUE;
145 }
```

```
147 DATATYPE Remove(LinkedList *list)
148 {
149     Node *temp = list->Curr;
150     DATATYPE tData = temp->Data;
151
152     list->Prev->next = list->Curr->next;
153     list->Curr = list->Prev;
154
155     free(temp);
156     list->NumofData--;
157     return tData;
158 }
159
160 int RetCount(LinkedList *list)
161 {
162     return list->NumofData;
163 }
164
```

SimpleGraph

```
165 //Graph
166 void InitGraph(Graph *G, int NumofVertex)
167 {
168     G->AdjList = (LinkedList*)malloc(sizeof(LinkedList) * NumofVertex);
169     G->NumofVertex = NumofVertex;
170     G->NumofEdge = 0;
171
172     for(int i = 0; i < NumofVertex; i++)
173     {
174         InitList(&G->AdjList[i]);
175     }
176 }
177
178 void ReleaseGraph(Graph* G)
179 {
180     if(G->AdjList != NULL)
181     {
182         free(G->AdjList);
183     }
184 }
```

SimpleGraph

```
186 void AddEdge(Graph *G, int From, int To)
187 {
188     Insert(&G->AdjList[From], To);
189     Insert(&G->AdjList[To], From);
190     G->NumofEdge++;
191 }
192
193 void PrintGraph(Graph *G)
194 {
195     int Vertex;
196
197     for(int i = 0; i < G->NumofVertex; i++)
198     {
199         printf("Vertex connected with %c : ", i + 65);
200
201         if(PosHead(&G->AdjList[i], &Vertex));
202         {
203             printf("%c ", Vertex + 65);
204
205             while(PosNext(&G->AdjList[i], &Vertex))
206             {
207                 printf("%c ", Vertex + 65);
208             }
209         }
210         printf("\n");
211     }
212 }
```

SimpleGraph

```
51 int main(int argc, char *argv[])
52 {
53     Graph graph;
54     InitGraph(&graph, 5);
55
56     AddEdge(&graph, A, B);
57     AddEdge(&graph, A, D);
58     AddEdge(&graph, B, C);
59     AddEdge(&graph, C, D);
60     AddEdge(&graph, D, E);
61     AddEdge(&graph, E, A);
62
63     PrintGraph(&graph);
64
65     ReleaseGraph(&graph);
66
67     return 0;
68 }
```

DFS ADT

- int VisitVertex(Graph *G, int Visit);
  - Marking vertex which visited and print out
- void PrintDFS(Graph *G, int Start);
  - Print vertex of G using depth first search

DFS

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TRUE 1
#define FALSE 0

#define STACKLEN 100

typedef int DATATYPE;

//stack
typedef struct
{
    DATATYPE StackArr[STACKLEN];
    int Top;
} ArrayStack;

//list
typedef struct Node{
    DATATYPE Data;
    struct Node *next;
} Node;

typedef struct LinkedList
{
    Node *Head;
    Node *Curr;
    Node *Prev;
    int NumofData;
} LinkedList;

//graph
enum {A, B, C, D, E, F, G, H, I, J};

typedef struct
{
    int NumofVertex;
    int NumofEdge;
    struct LinkedList *AdjList;
    int *VisitInfo;
} Graph;
```

DFS

```
52 //list
53 void InitList(LinkedList *list);
54 void Insert(LinkedList *list, DATATYPE Data);
55
56 void HeadInsert(LinkedList *list, DATATYPE Data);
57 void SortInsert(LinkedList *list, DATATYPE Data);
58
59 int PosHead(LinkedList *list, DATATYPE *Data);
60 int PosNext(LinkedList *list, DATATYPE *Data);
61
62 DATATYPE Remove(LinkedList *list);
63 int RetCount(LinkedList *list);
```

```
65 //graph
66 void InitGraph(Graph *G, int NumofVertex);
67 void ReleaseGraph(Graph* G);
68
69 void AddEdge(Graph *G, int From, int To);
70 void PrintGraph(Graph *G);
71
72 int VisitVertex(Graph *G, int Visit);
73 void PrintDFS(Graph *G, int Start);
```

DFS

```
102 //stack
103 void InitStack(ArrayStack *stack)
104 {
105     stack->Top = -1;
106 }
107
108 int IsEmpty(ArrayStack *stack)
109 {
110     if(stack->Top == -1)
111     {
112         return TRUE;
113     }
114     else
115     {
116         return FALSE;
117     }
118 }
119
120 void Push(ArrayStack *stack, DATATYPE data)
121 {
122     stack->Top += 1;
123     stack->StackArr[stack->Top] = data;
124 }
```

```
126 DATATYPE Pop(ArrayStack *stack)
127 {
128     int tempIdx;
129
130     if(IsEmpty(stack))
131     {
132         printf("Stack is empty\n");
133         exit(1);
134     }
135
136     tempIdx = stack->Top;
137     stack->Top -= 1;
138
139     return stack->StackArr[tempIdx];
140 }
141
142 DATATYPE Peek(ArrayStack *stack)
143 {
144     if(IsEmpty(stack))
145     {
146         printf("Stack is empty\n");
147         exit(1);
148     }
149
150     return stack->StackArr[stack->Top];
151 }
```

DFS

```
153 //list
154 void InitList(LinkedList *list)
155 {
156     list->Head = (Node*)malloc(sizeof(Node));
157     list->Head->next = NULL;
158     list->NumofData = 0;
159 }
160
161 void Insert(LinkedList *list, DATATYPE Data)
162 {
163     if(list->Head->next == NULL)
164     {
165         HeadInsert(list, Data);
166     }
167     else
168     {
169         SortInsert(list, Data);
170     }
171 }
172
173 void HeadInsert(LinkedList *list, DATATYPE Data)
174 {
175     Node *temp = (Node*)malloc(sizeof(Node));
176     temp->Data = Data;
177
178     temp->next = list->Head->next;
179     list->Head->next = temp;
180
181     list->NumofData++;
182 }
```

```
184 void SortInsert(LinkedList *list, DATATYPE Data)
185 {
186     Node *new = (Node*)malloc(sizeof(Node));
187     Node *pred = list->Head;
188     new->Data = Data;
189
190     //find pos
191     while((pred->next != NULL) && (Data > pred->next->Data))
192     {
193         pred = pred->next;
194     }
195
196     new->next = pred->next;
197     pred->next = new;
198
199     list->NumofData++;
200 }
201
202 int PosHead(LinkedList *list, DATATYPE *Data)
203 {
204     if(list->Head->next == NULL)
205     {
206         return FALSE;
207     }
208
209     list->Prev = list->Head;
210     list->Curr = list->Head->next;
211
212     *Data = list->Curr->Data;
213     return TRUE;
214 }
215
216 int PosNext(LinkedList *list, DATATYPE *Data)
217 {
218     if(list->Curr->next == NULL)
219     {
220         return FALSE;
221     }
222
223     list->Prev = list->Curr;
224     list->Curr = list->Curr->next;
225
226     *Data = list->Curr->Data;
227     return TRUE;
228 }
```

DFS

```
230 DATATYPE Remove(LinkedList *list)
231 {
232     Node *temp = list->Curr;
233     DATATYPE tData = temp->Data;
234
235     list->Prev->next = list->Curr->next;
236     list->Curr = list->Prev;
237
238     free(temp);
239     list->NumofData--;
240     return tData;
241 }
242
243 int RetCount(LinkedList *list)
244 {
245     return list->NumofData;
246 }
247
248 //Graph
249 void InitGraph(Graph *G, int NumofVertex)
250 {
251     G->AdjList = (LinkedList*)malloc(sizeof(LinkedList) * NumofVertex);
252     G->NumofVertex = NumofVertex;
253     G->NumofEdge = 0;
254
255     for(int i = 0; i < NumofVertex; i++)
256     {
257         InitList(&G->AdjList[i]);
258     }
259
260     G->VisitInfo = (int*)malloc(sizeof(int) * NumofVertex);
261     for(int i = 0; i < NumofVertex; i++)
262     {
263         G->VisitInfo[i] = 0;
264     }
265 }
```

DFS

```
267 void ReleaseGraph(Graph* G)
268 {
269     if(G->AdjList != NULL)
270     {
271         free(G->AdjList);
272     }
273
274     if(G->VisitInfo != NULL)
275     {
276         free(G->VisitInfo);
277     }
278 }
279
280 void AddEdge(Graph *G, int From, int To)
281 {
282     Insert(&G->AdjList[From], To);
283     Insert(&G->AdjList[To], From);
284     G->NumofEdge++;
285 }
286
287 void PrintGraph(Graph *G)
288 {
289     int Vertex;
290
291     for(int i = 0; i < G->NumofVertex; i++)
292     {
293         printf("Vertex connected with %c : ", i + 65);
294
295         if(PosHead(&G->AdjList[i], &Vertex));
296         {
297             printf("%c ", Vertex + 65);
298
299             while(PosNext(&G->AdjList[i], &Vertex))
300             {
301                 printf("%c ", Vertex + 65);
302             }
303         }
304         printf("\n");
305     }
306 }
```

```
308 int VisitVertex(Graph *G, int Visit)
309 {
310     if(G->VisitInfo[Visit] == 0)
311     {
312         G->VisitInfo[Visit] = 1;
313         printf("%c ", Visit + 65);
314         return TRUE;
315     }
316     return FALSE;
317 }
```

```
319 void PrintDFS(Graph *G, int Start)
320 {
321     ArrayStack stack;
322     int Visit = Start;
323     int Next;
324 
325     InitStack(&stack);
326     VisitVertex(G, Visit);
327     Push(&stack, Visit);
328 
329     while(PosHead(&G->AdjList[Visit], &Next))
330     {
331         int Flag = FALSE;
332 
333         if(VisitVertex(G, Next))
334         {
335             Push(&stack, Visit);
336             Visit = Next;
337             Flag = TRUE;
338         }
339         else
340         {
341             while(PosNext(&G->AdjList[Visit], &Next))
342             {
343                 if(VisitVertex(G, Next))
344                 {
345                     Push(&stack, Visit);
346                     Visit = Next;
347                     Flag = TRUE;
348                     break;
349                 }
350             }
351         }
352 
353         if(Flag == FALSE)
354         {
355             if(IsEmpty(&stack))
356             {
357                 break;
358             }
359             else
360             {
361                 Visit = Pop(&stack);
362             }
363         }
364     }
365 
366     for(int i = 0; i < G->NumofVertex; i++)
367     {
368         G->VisitInfo[i] = 0;
369     }
370 }
```

DFS

```
75 int main(int argc, char *argv[])
76 {
77     Graph graph;
78
79     InitGraph(&graph, 5);
80     AddEdge(&graph, A, B);
81     AddEdge(&graph, A, D);
82     AddEdge(&graph, B, C);
83     AddEdge(&graph, C, D);
84     AddEdge(&graph, D, E);
85     AddEdge(&graph, E, A);
86
87     PrintGraph(&graph);
88     printf("\n");
89
90     PrintDFS(&graph, A);
91     printf("\n");
92     PrintDFS(&graph, D);
93     printf("\n");
94     PrintDFS(&graph, B);
95     printf("\n");
96
97     ReleaseGraph(&graph);
98
99     return 0;
100 }
```

Lab10: Graph

- Submit on Blackboard
- Capture the output of the DFS program under given condisions
- Submit file : screenshots of output(image file, 3)
- Filename : 1, 2, 3(with .jpg, .png, etc)

Lab10: Graph

- Main condition : 3 random graphs with 7 or more vertices
  - Sub condition 1 : graph with no cycles
  - Sub condition 2 : graph has cycle with 3 or fewer vertices
  - Sub condition 3 : graph has cycle with 5 or more vertices
- Take screenshot of the graphs that satisfies the main condition and each sub condition

Lab10: Graph

- Note : do not change any codes