HYU HANYANG UNIVERSITY DIVISION OF COMPUTER SCIENCE

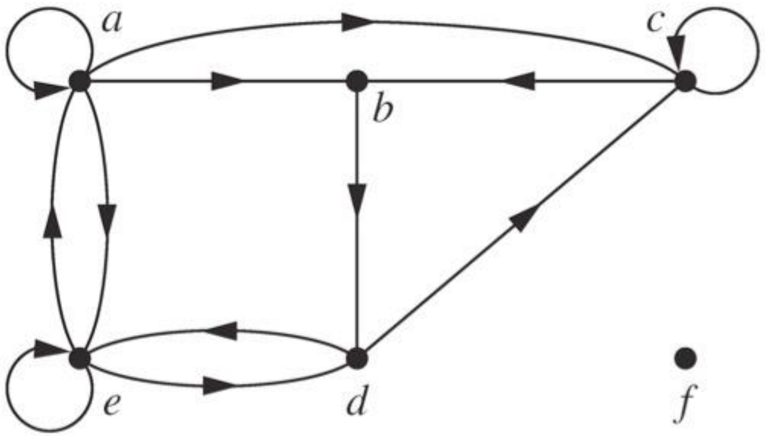
Data Structures

박영준 교수님

Lab10: Graph

Graph

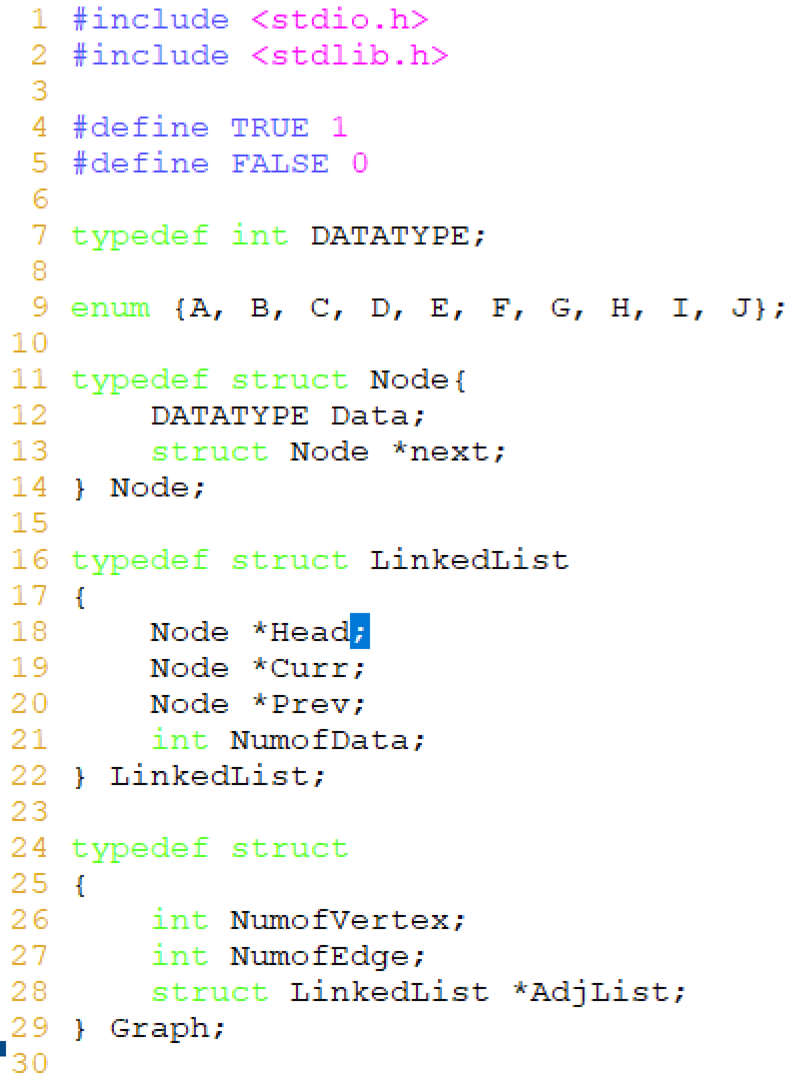
- Non-linear data structure consisting of nodes(vertex) and edges.



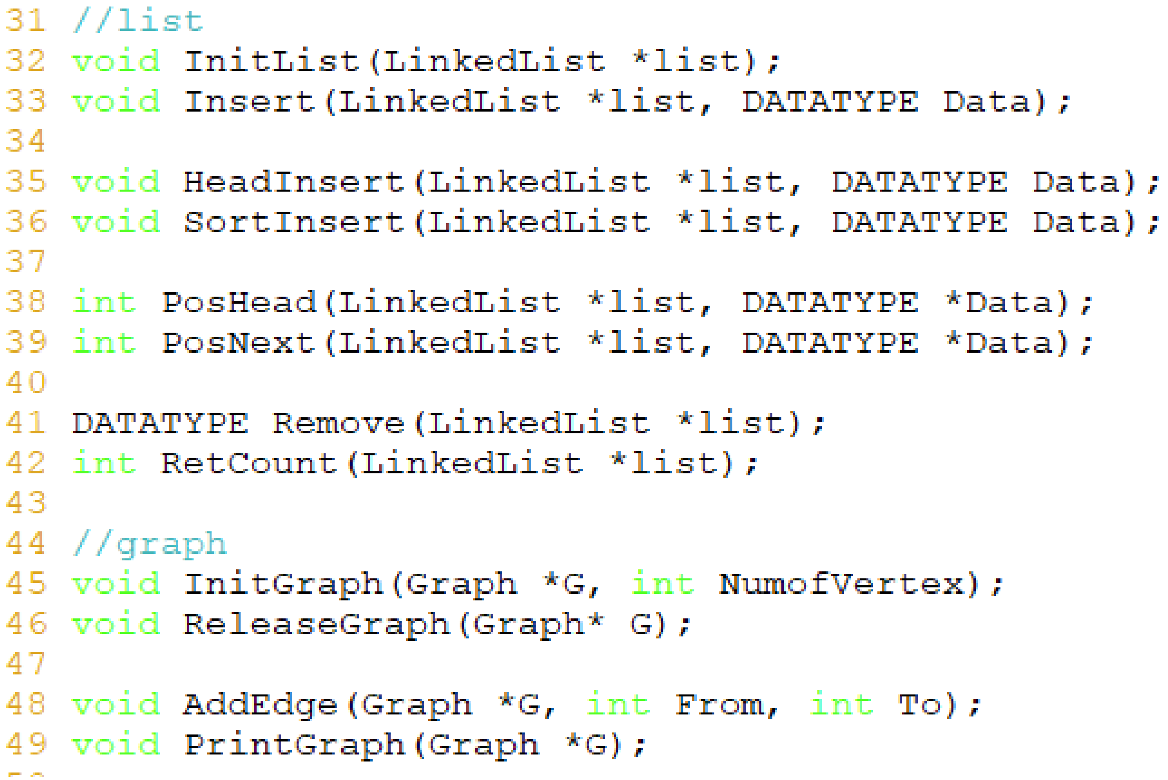
SimpleGraph ADT

- 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

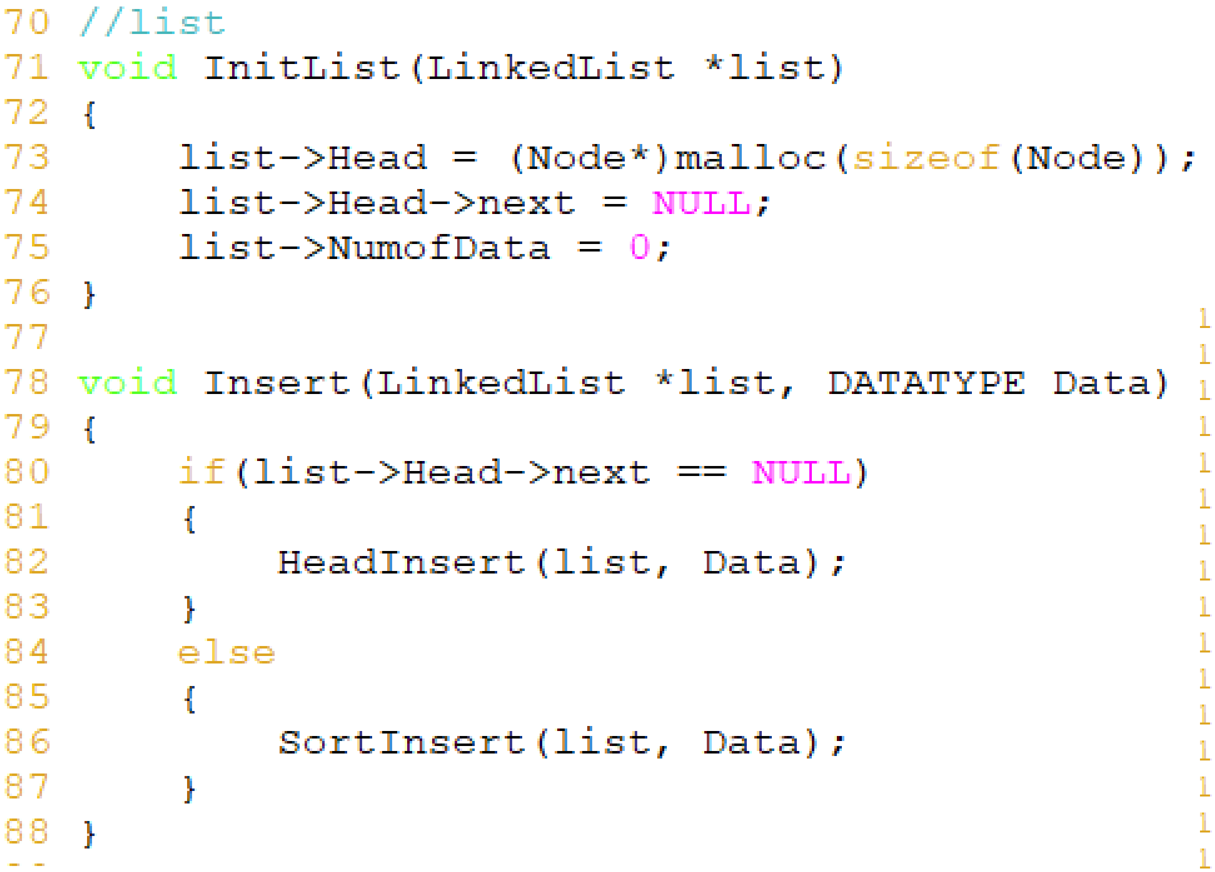


```c
#include <stdio.h>
#include <stdlib.h>

#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;

```

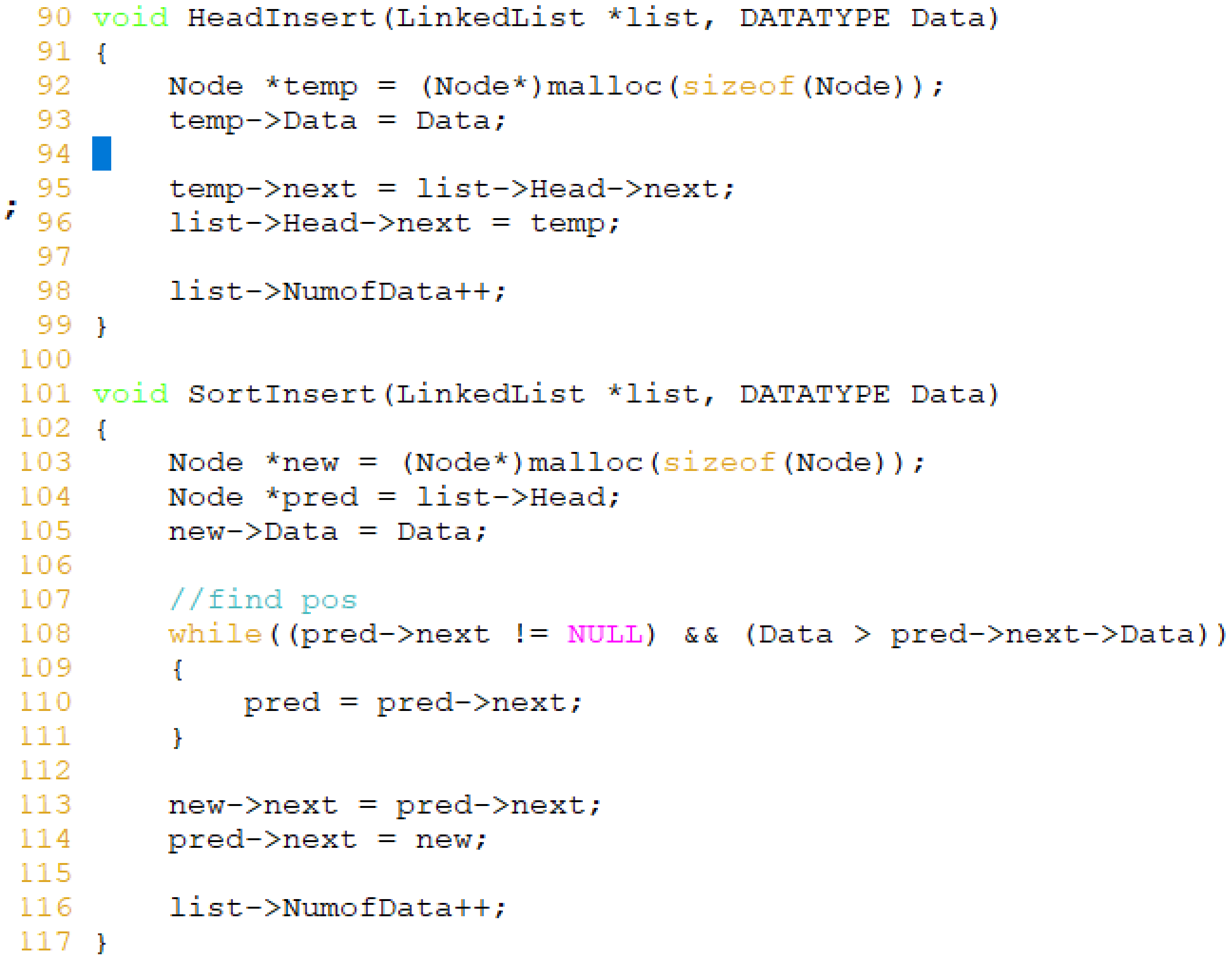


```c
//list
void InitList(LinkedList *list);
void Insert(LinkedList *list, DATATYPE Data);

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

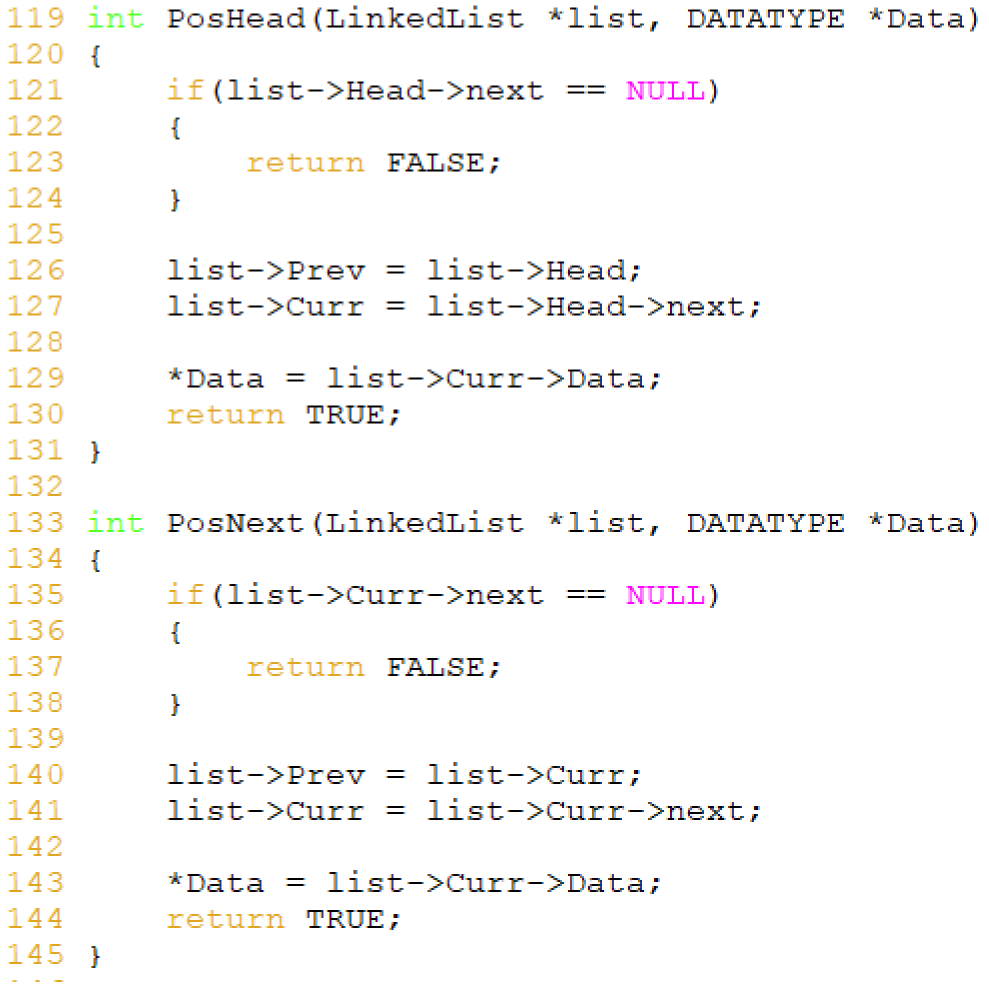


```c
//list
void InitList(LinkedList *list)
{
    list->Head = (Node*)malloc(sizeof(Node));
    list->Head->next = NULL;
    list->NumofData = 0;
}

void Insert(LinkedList *list, DATATYPE Data)
{
    if(list->Head->next == NULL)
    {
        HeadInsert(list, Data);
    }
    else
    {
        SortInsert(list, Data);
    }
}
```

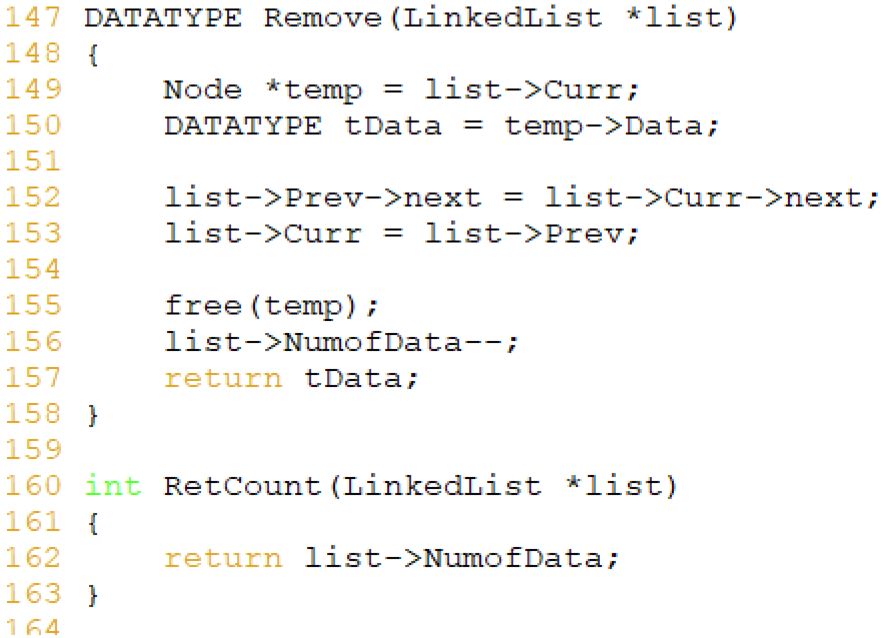


```c
void HeadInsert(LinkedList *list, DATATYPE Data)
{
    Node *temp = (Node*)malloc(sizeof(Node));
    temp->Data = Data;

    temp->next = list->Head->next;
    list->Head->next = temp;

    list->NumofData++;
}

void SortInsert(LinkedList *list, DATATYPE Data)
{
    Node *new = (Node*)malloc(sizeof(Node));
    Node *pred = list->Head;
    new->Data = Data;

    //find pos
    while((pred->next != NULL) && (Data > pred->next->Data))
    {
        pred = pred->next;
    }

    new->next = pred->next;
    pred->next = new;

    list->NumofData++;
}
```

SimpleGraph

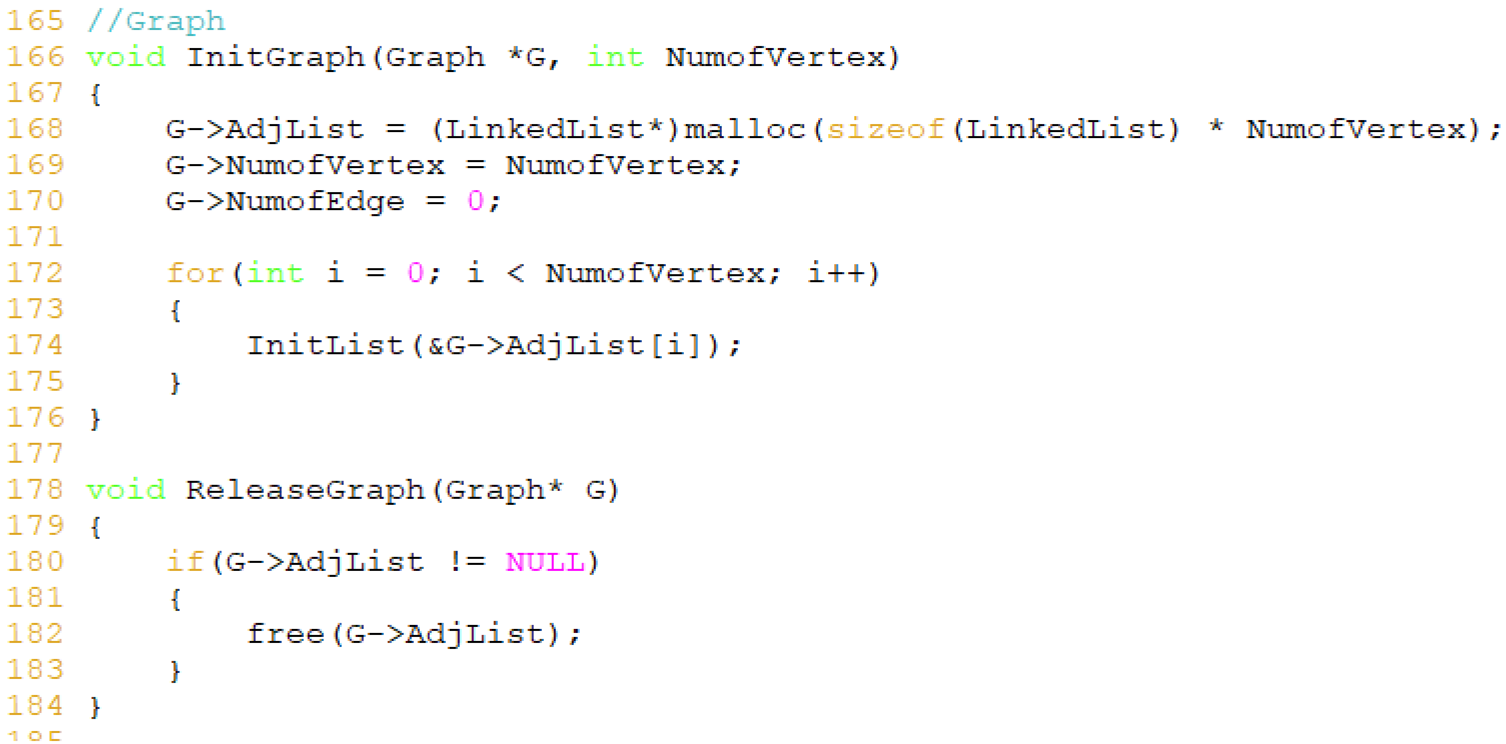


```c
int PosHead(LinkedList *list, DATATYPE *Data)
{
    if(list->Head->next == NULL)
    {
        return FALSE;
    }

    list->Prev = list->Head;
    list->Curr = list->Head->next;

    *Data = list->Curr->Data;
    return TRUE;
}

int PosNext(LinkedList *list, DATATYPE *Data)
{
    if(list->Curr->next == NULL)
    {
        return FALSE;
    }

    list->Prev = list->Curr;
    list->Curr = list->Curr->next;

    *Data = list->Curr->Data;
    return TRUE;
}
```



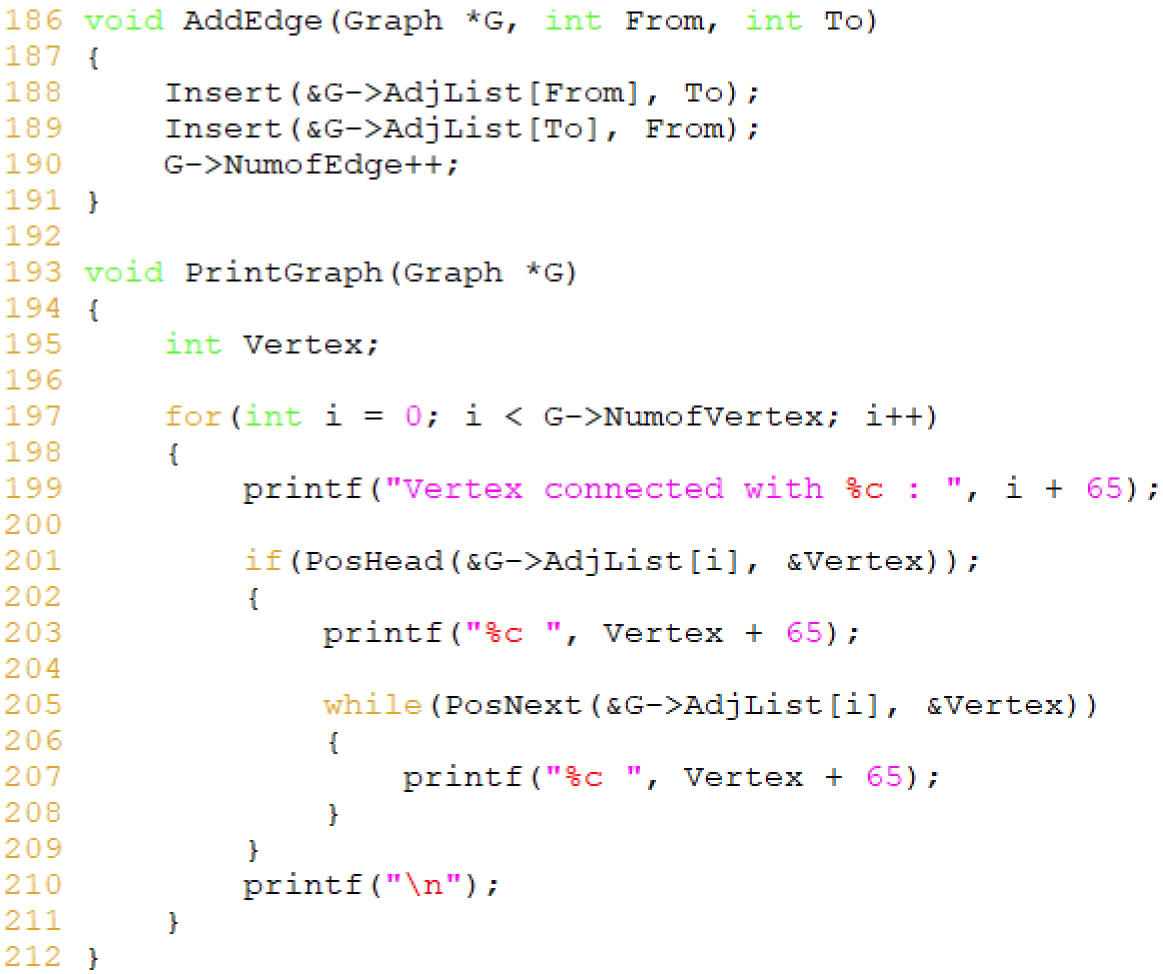
```c
DATATYPE Remove(LinkedList *list)
{
    Node *temp = list->Curr;
    DATATYPE tData = temp->Data;

    list->Prev->next = list->Curr->next;
    list->Curr = list->Prev;

    free(temp);
    list->NumofData--;
    return tData;
}

int RetCount(LinkedList *list)
{
    return list->NumofData;
}
```

SimpleGraph



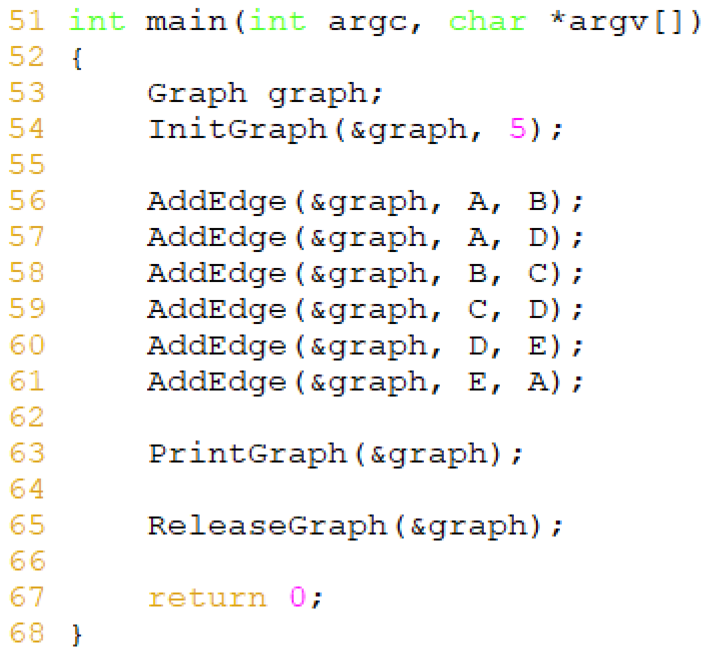
```
//Graph
void InitGraph(Graph *G, int NumofVertex)
{
    G->AdjList = (LinkedList*)malloc(sizeof(LinkedList) * NumofVertex);
    G->NumofVertex = NumofVertex;
    G->NumofEdge = 0;

    for(int i = 0; i < NumofVertex; i++)
    {
        InitList(&G->AdjList[i]);
    }
}

void ReleaseGraph(Graph* G)
{
    if(G->AdjList != NULL)
    {
        free(G->AdjList);
    }
}
```

SimpleGraph



```c
void AddEdge(Graph *G, int From, int To)
{
    Insert(&G->AdjList[From], To);
    Insert(&G->AdjList[To], From);
    G->NumofEdge++;
}

void PrintGraph(Graph *G)
{
    int Vertex;

    for(int i = 0; i < G->NumofVertex; i++)
    {
        printf("Vertex connected with %c : ", i + 65);

        if(PosHead(&G->AdjList[i], &Vertex));
        {
            printf("%c ", Vertex + 65);

            while(PosNext(&G->AdjList[i], &Vertex))
            {
                printf("%c ", Vertex + 65);
            }
        }
        printf("\n");
    }
}
```

SimpleGraph

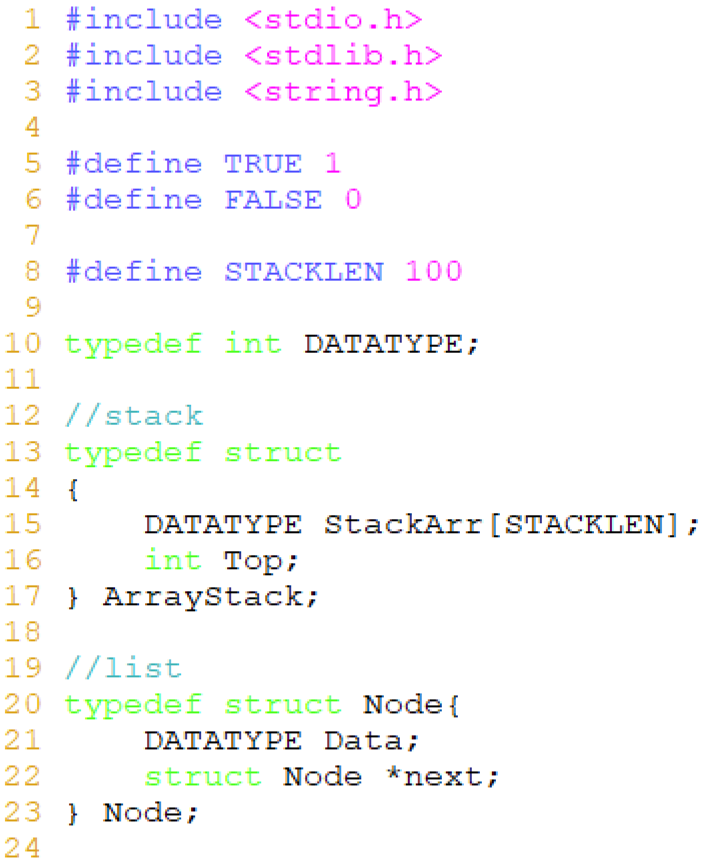


```c
int main(int argc, char *argv[])
{
    Graph graph;
    InitGraph(&graph, 5);

    AddEdge(&graph, A, B);
    AddEdge(&graph, A, D);
    AddEdge(&graph, B, C);
    AddEdge(&graph, C, D);
    AddEdge(&graph, D, E);
    AddEdge(&graph, E, A);

    PrintGraph(&graph);

    ReleaseGraph(&graph);

    return 0;
}
```

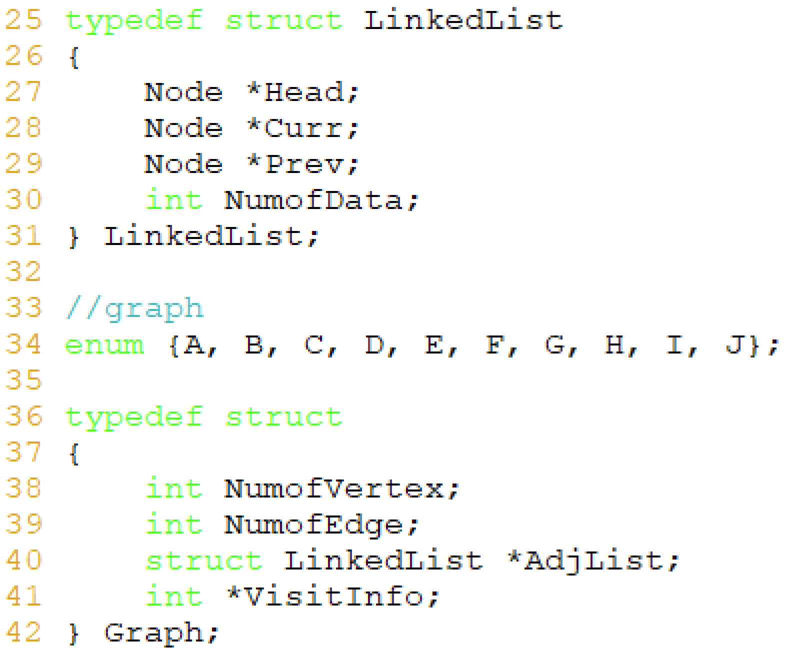
DFS ADT

- 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

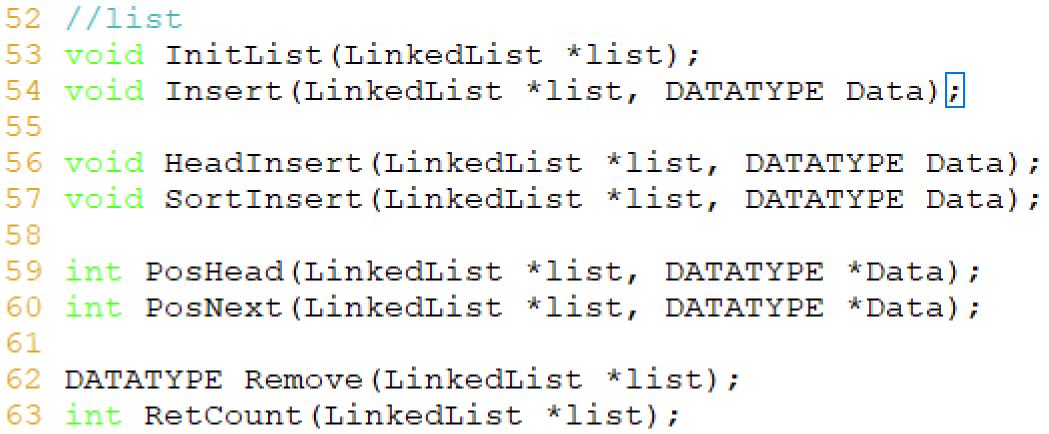


```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#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;

```

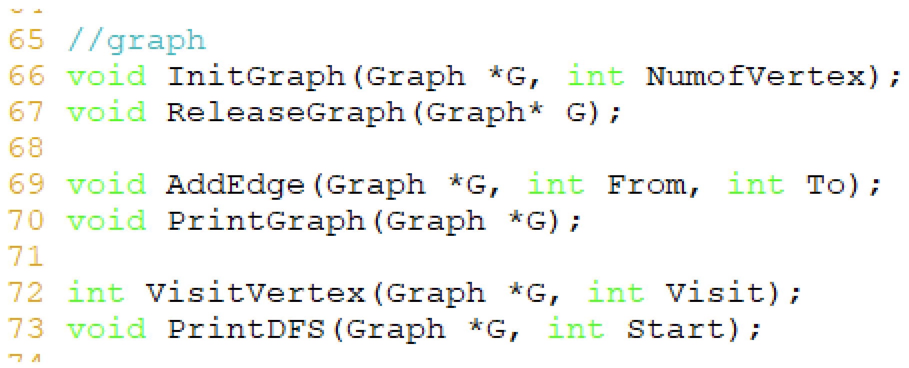


```c
typedef struct LinkedList
{
    Node *Head;
    Node *Curr;
    Node *Prev;
    int NumofData;
} LinkedList;

//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

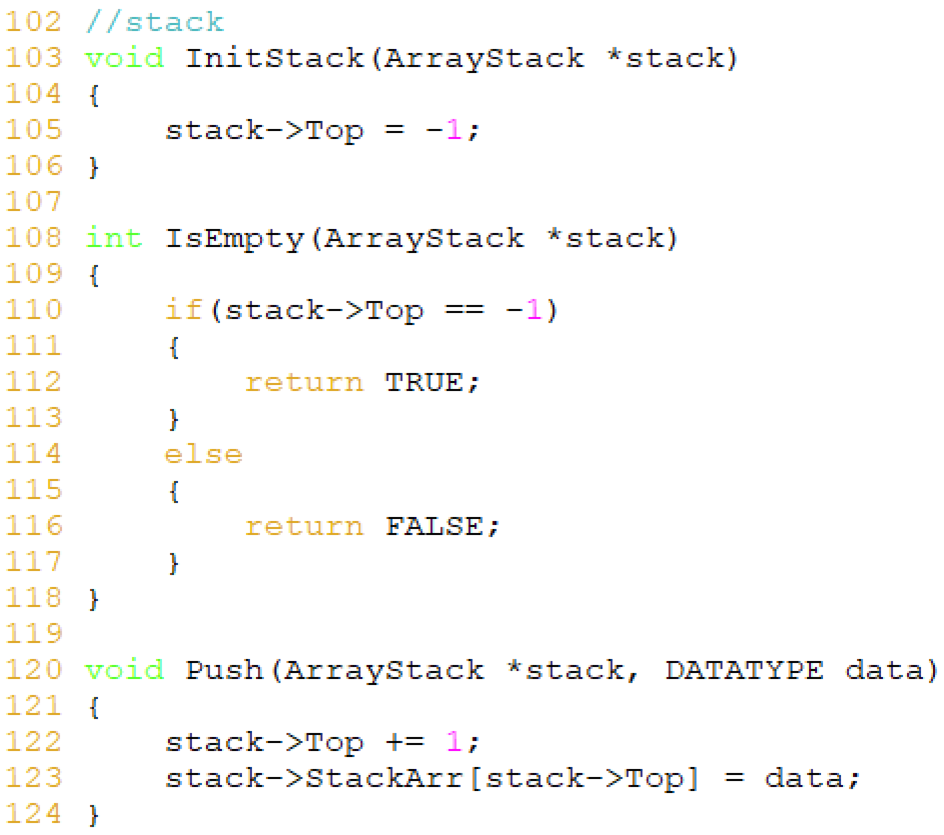


```c
//list
void InitList(LinkedList *list);
void Insert(LinkedList *list, DATATYPE Data);

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);
```

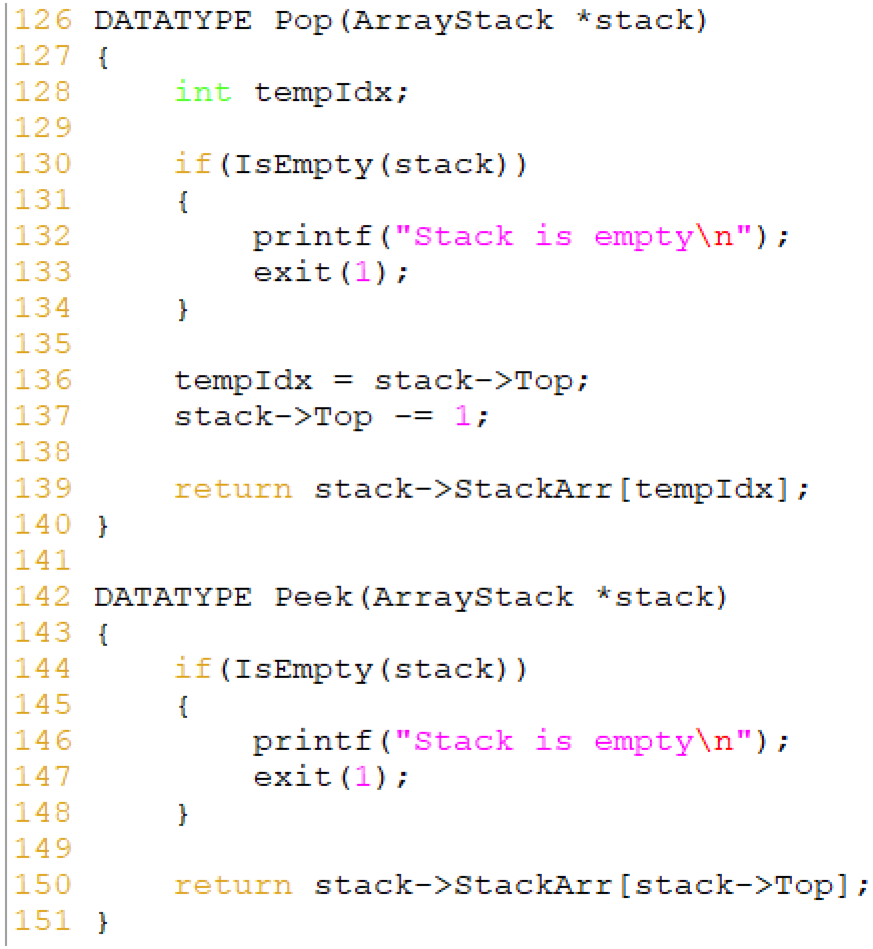


```c
//graph
void InitGraph(Graph *G, int NumofVertex);
void ReleaseGraph(Graph* G);

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

int VisitVertex(Graph *G, int Visit);
void PrintDFS(Graph *G, int Start);
```

DFS

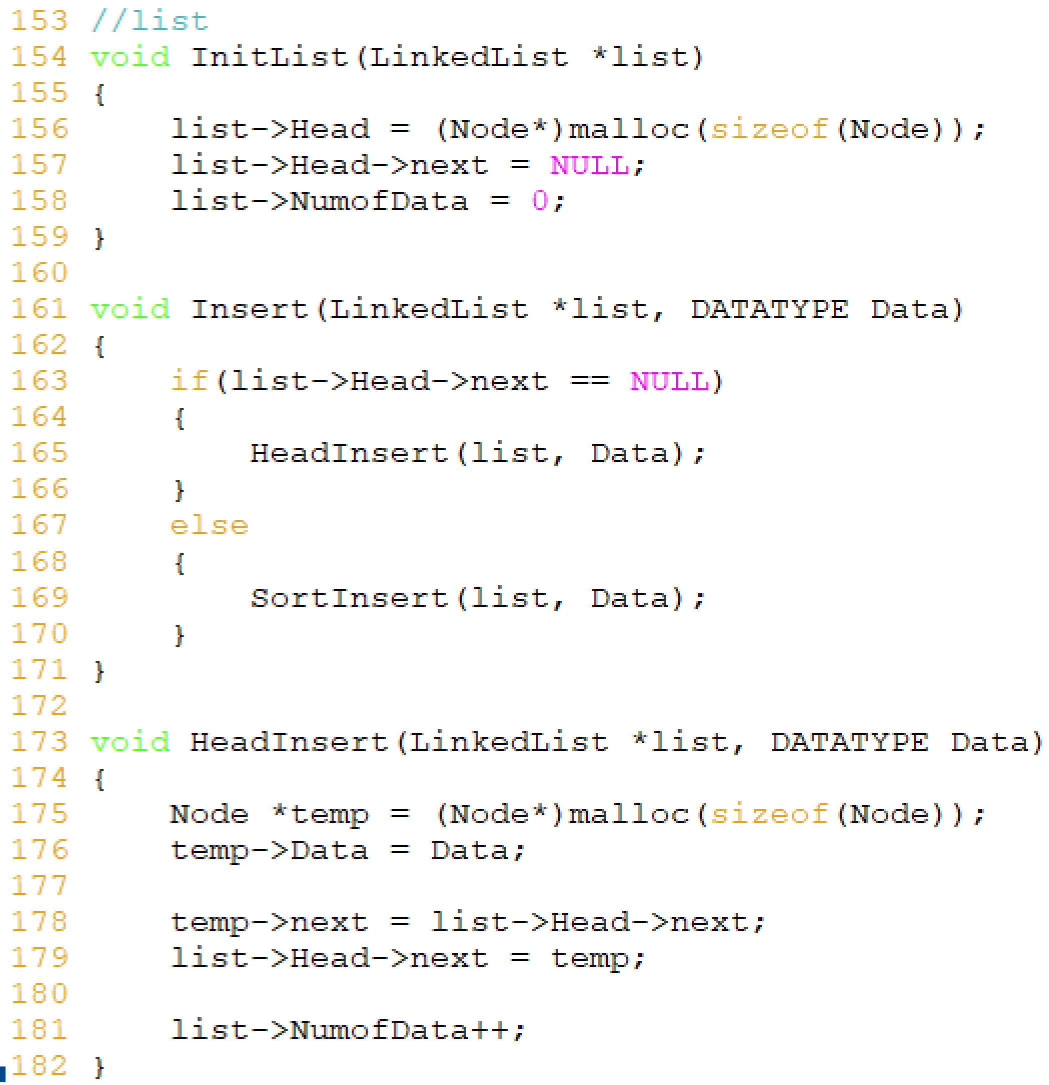


```c
//stack
void InitStack(ArrayStack *stack)
{
    stack->Top = -1;
}

int IsEmpty(ArrayStack *stack)
{
    if(stack->Top == -1)
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}

void Push(ArrayStack *stack, DATATYPE data)
{
    stack->Top += 1;
    stack->StackArr[stack->Top] = data;
}
```

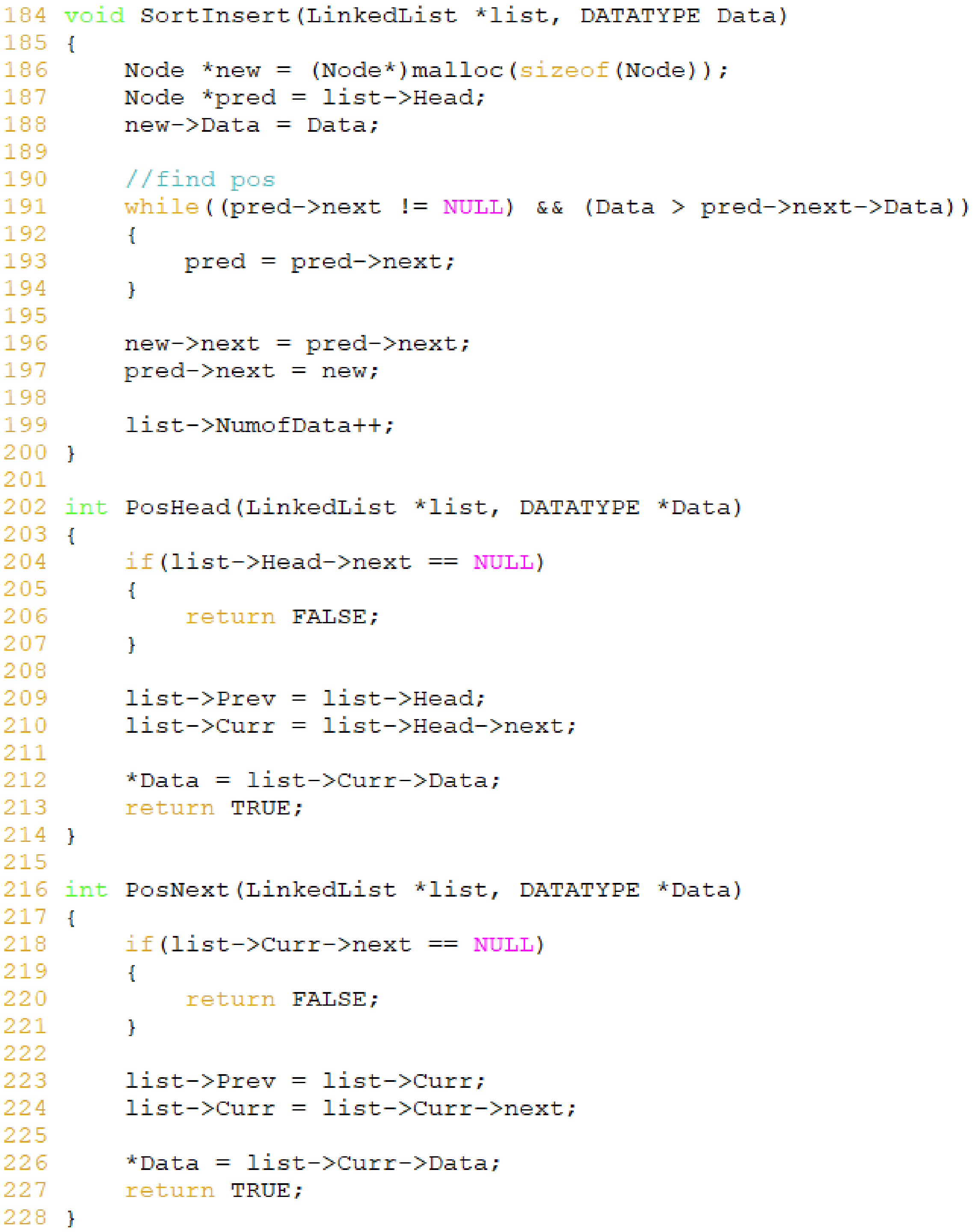


```c
DATATYPE Pop(ArrayStack *stack)
{
    int tempIdx;

    if(IsEmpty(stack))
    {
        printf("Stack is empty\n");
        exit(1);
    }

    tempIdx = stack->Top;
    stack->Top -= 1;

    return stack->StackArr[tempIdx];
}

DATATYPE Peek(ArrayStack *stack)
{
    if(IsEmpty(stack))
    {
        printf("Stack is empty\n");
        exit(1);
    }

    return stack->StackArr[stack->Top];
}
```

DFS

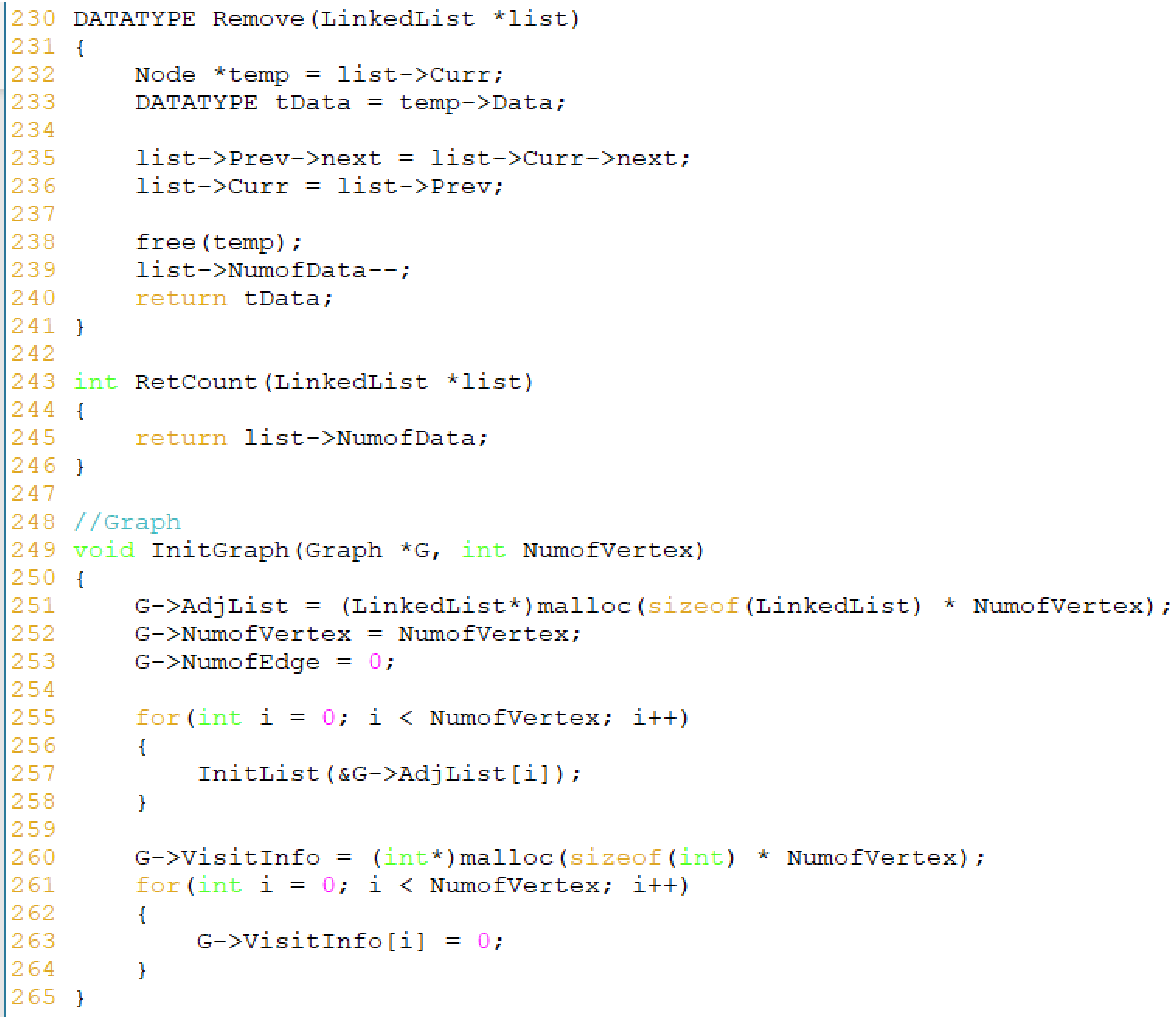


```c
//list
void InitList(LinkedList *list)
{
    list->Head = (Node*)malloc(sizeof(Node));
    list->Head->next = NULL;
    list->NumofData = 0;
}

void Insert(LinkedList *list, DATATYPE Data)
{
    if(list->Head->next == NULL)
    {
        HeadInsert(list, Data);
    }
    else
    {
        SortInsert(list, Data);
    }
}

void HeadInsert(LinkedList *list, DATATYPE Data)
{
    Node *temp = (Node*)malloc(sizeof(Node));
    temp->Data = Data;

    temp->next = list->Head->next;
    list->Head->next = temp;

    list->NumofData++;
}
```



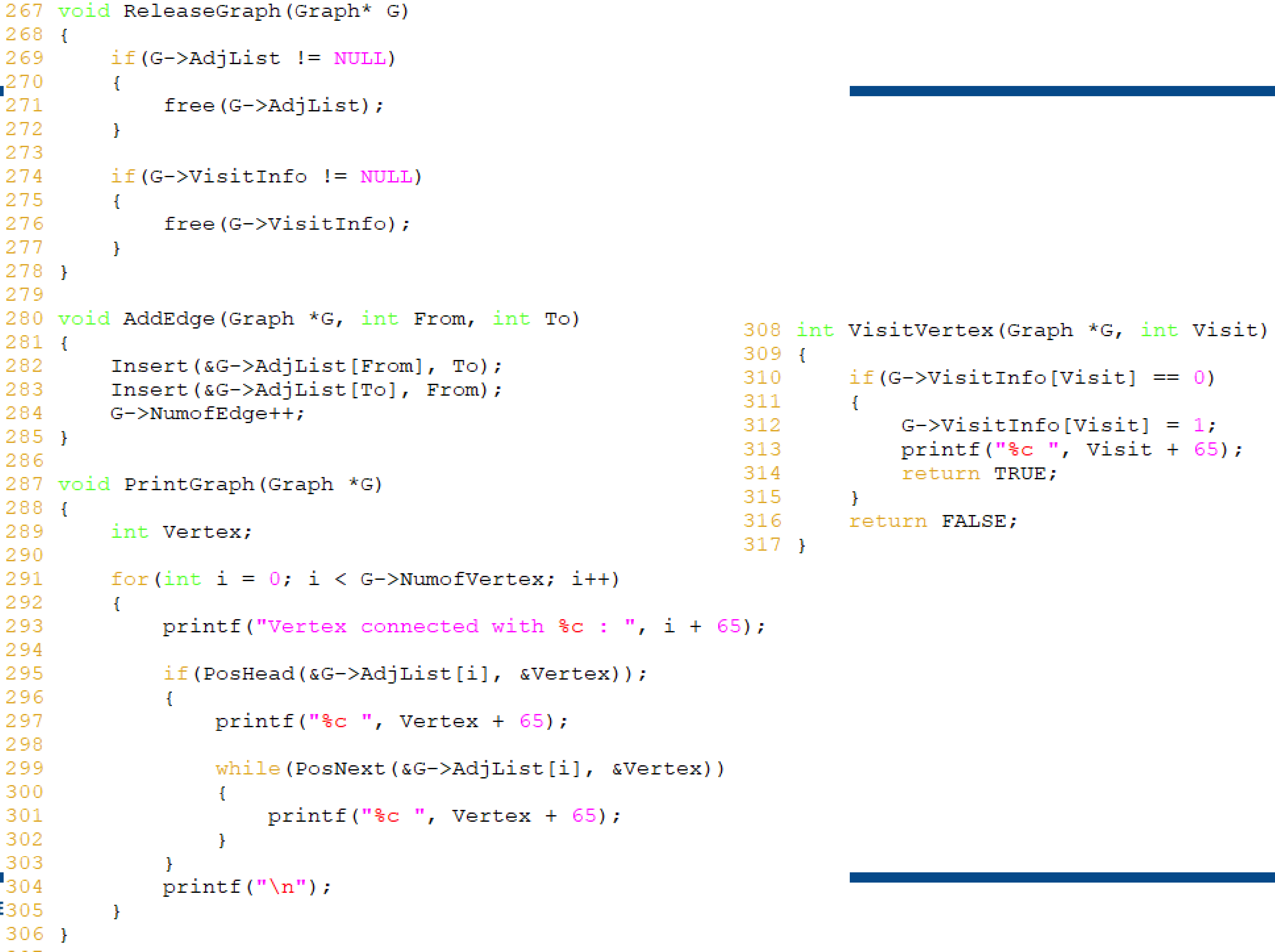
```c
void SortInsert(LinkedList *list, DATATYPE Data)
{
    Node *new = (Node*)malloc(sizeof(Node));
    Node *pred = list->Head;
    new->Data = Data;

    //find pos
    while((pred->next != NULL) && (Data > pred->next->Data))
    {
        pred = pred->next;
    }

    new->next = pred->next;
    pred->next = new;

    list->NumofData++;
}

int PosHead(LinkedList *list, DATATYPE *Data)
{
    if(list->Head->next == NULL)
    {
        return FALSE;
    }

    list->Prev = list->Head;
    list->Curr = list->Head->next;

    *Data = list->Curr->Data;
    return TRUE;
}

int PosNext(LinkedList *list, DATATYPE *Data)
{
    if(list->Curr->next == NULL)
    {
        return FALSE;
    }

    list->Prev = list->Curr;
    list->Curr = list->Curr->next;

    *Data = list->Curr->Data;
    return TRUE;
}
```

DFS



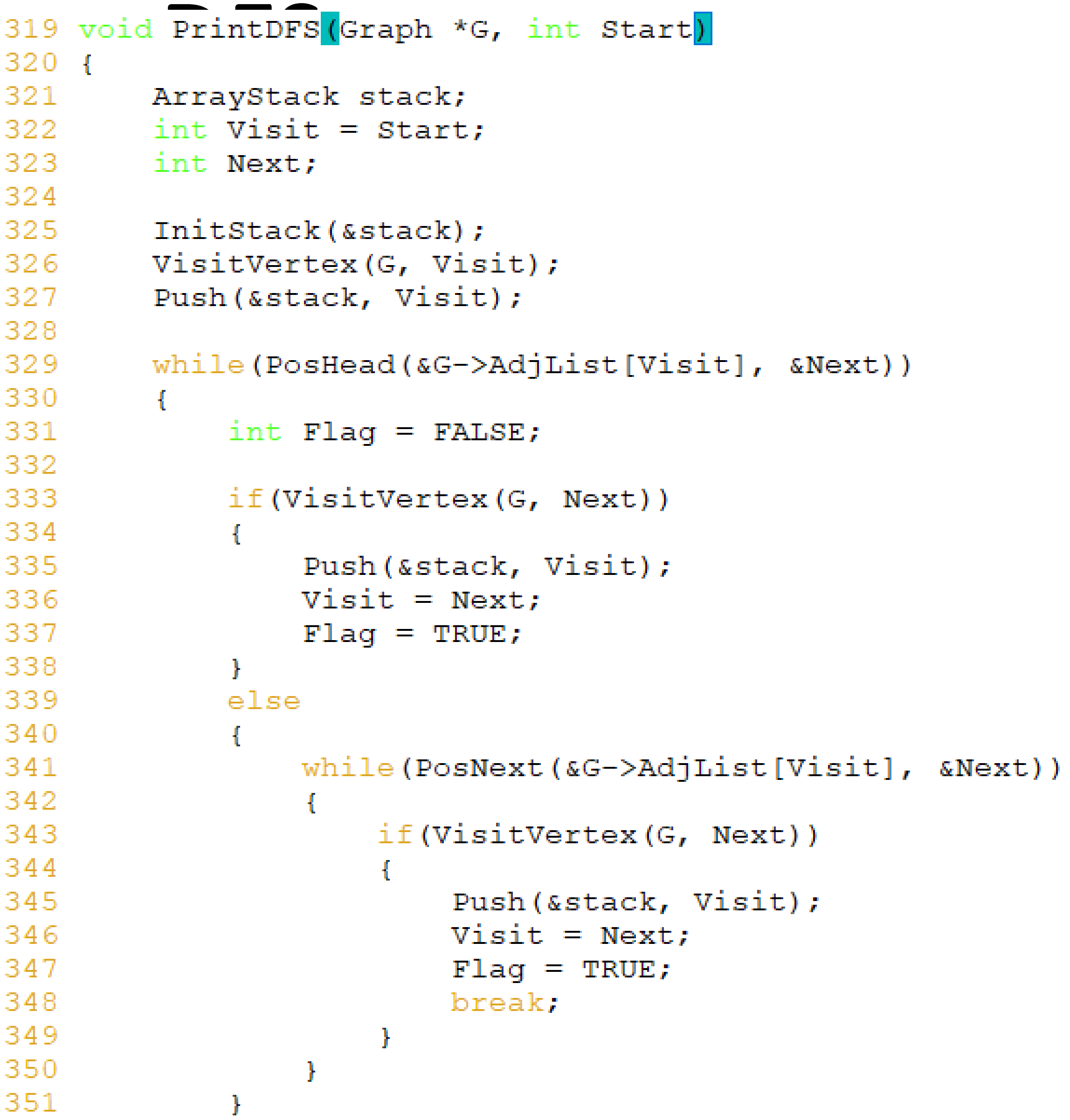
```
DATATYPE Remove(LinkedList *list)
{
    Node *temp = list->Curr;
    DATATYPE tData = temp->Data;

    list->Prev->next = list->Curr->next;
    list->Curr = list->Prev;

    free(temp);
    list->NumofData--;
    return tData;
}

int RetCount(LinkedList *list)
{
    return list->NumofData;
}

//Graph
void InitGraph(Graph *G, int NumofVertex)
{
    G->AdjList = (LinkedList*)malloc(sizeof(LinkedList) * NumofVertex);
    G->NumofVertex = NumofVertex;
    G->NumofEdge = 0;

    for(int i = 0; i < NumofVertex; i++)
    {
        InitList(&G->AdjList[i]);
    }

    G->VisitInfo = (int*)malloc(sizeof(int) * NumofVertex);
    for(int i = 0; i < NumofVertex; i++)
    {
        G->VisitInfo[i] = 0;
    }
}
```

DFS

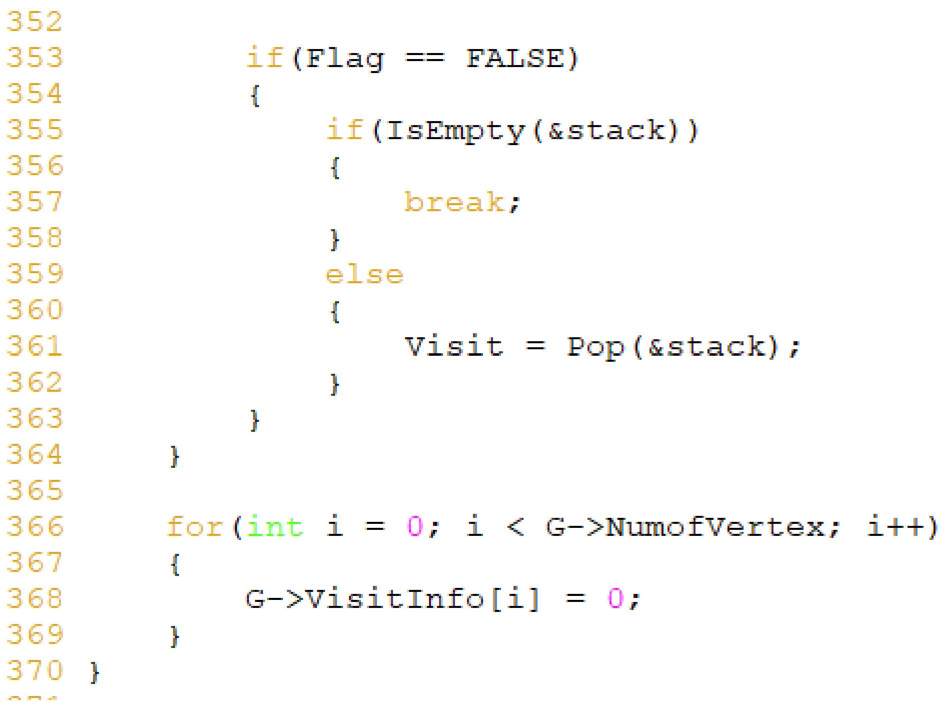


```c
void ReleaseGraph(Graph* G)
{
    if(G->AdjList != NULL)
    {
        free(G->AdjList);
    }

    if(G->VisitInfo != NULL)
    {
        free(G->VisitInfo);
    }
}

void AddEdge(Graph *G, int From, int To)
{
    Insert(&G->AdjList[From], To);
    Insert(&G->AdjList[To], From);
    G->NumofEdge++;
}

void PrintGraph(Graph *G)
{
    int Vertex;

    for(int i = 0; i < G->NumofVertex; i++)
    {
        printf("Vertex connected with %c : ", i + 65);

        if(PosHead(&G->AdjList[i], &Vertex));
        {
            printf("%c ", Vertex + 65);

            while(PosNext(&G->AdjList[i], &Vertex))
            {
                printf("%c ", Vertex + 65);
            }
        }
        printf("\n");
    }
}
```

```c
int VisitVertex(Graph *G, int Visit)
{
    if(G->VisitInfo[Visit] == 0)
    {
        G->VisitInfo[Visit] = 1;
        printf("%c ", Visit + 65);
        return TRUE;
    }
    return FALSE;
}
```

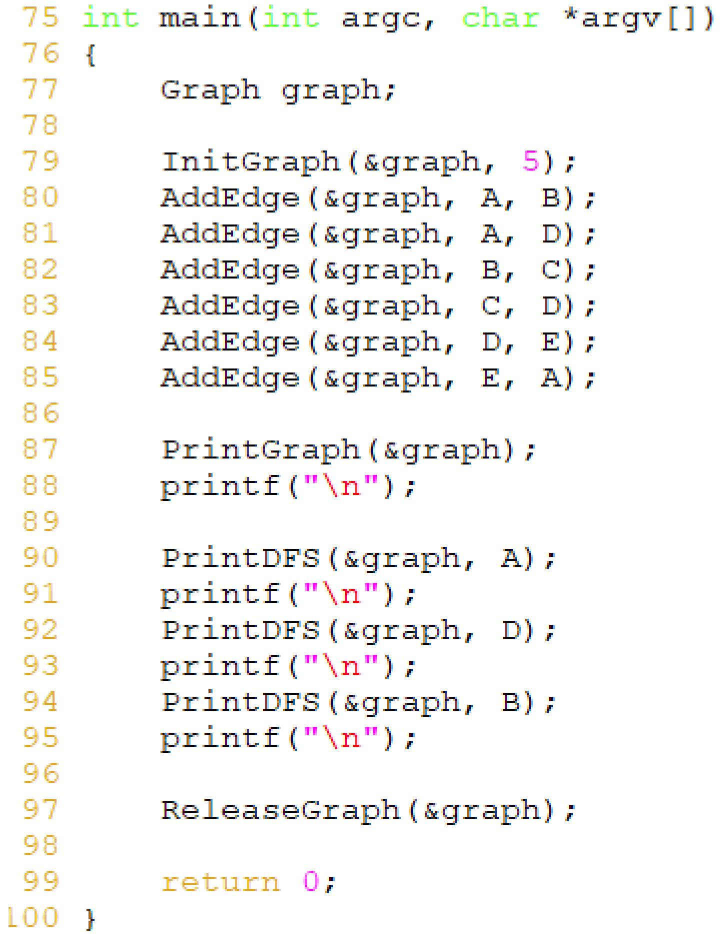


```c
void PrintDFS(Graph *G, int Start)
{
    ArrayStack stack;
    int Visit = Start;
    int Next;

    InitStack(&stack);
    VisitVertex(G, Visit);
    Push(&stack, Visit);

    while(PosHead(&G->AdjList[Visit], &Next))
    {
        int Flag = FALSE;

        if(VisitVertex(G, Next))
        {
            Push(&stack, Visit);
            Visit = Next;
            Flag = TRUE;
        }
        else
        {
            while(PosNext(&G->AdjList[Visit], &Next))
            {
                if(VisitVertex(G, Next))
                {
                    Push(&stack, Visit);
                    Visit = Next;
                    Flag = TRUE;
                    break;
                }
            }
        }
```



```c

        if(Flag == FALSE)
        {
            if(IsEmpty(&stack))
            {
                break;
            }
            else
            {
                Visit = Pop(&stack);
            }
        }
    }

    for(int i = 0; i < G->NumofVertex; i++)
    {
        G->VisitInfo[i] = 0;
    }
}
```

DFS



```c
int main(int argc, char *argv[])
{
    Graph graph;

    InitGraph(&graph, 5);
    AddEdge(&graph, A, B);
    AddEdge(&graph, A, D);
    AddEdge(&graph, B, C);
    AddEdge(&graph, C, D);
    AddEdge(&graph, D, E);
    AddEdge(&graph, E, A);

    PrintGraph(&graph);
    printf("\n");

    PrintDFS(&graph, A);
    printf("\n");
    PrintDFS(&graph, D);
    printf("\n");
    PrintDFS(&graph, B);
    printf("\n");

    ReleaseGraph(&graph);

    return 0;
}
```

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